République Algérienne Démocratique et Populaire

الجمهوريـــــة الجزائريـــــة الديمقراطيـــــة الشعبيـــــة

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

وزارة التعليم العالي و البحث العلمي

Université 08 Mai 1945 Guelma

جامعة 8 ماي1945 قالمة

Faculté des Mathématiques, de l'Informatique Et des Sciences de la matière

كلية الرياضيات ، الإعلام الآلي وعلـــــوم المـــــادة

Département d'informatique

قسم الاعلام الآلـــــي

Polycopié

Logique et Fondements de l'Informatique 1

Cours et Exercices Corrigés

Parcours: Master Systèmes Informatiques (SI)
1ière Année - Semestre 1

Réalisé par: Dr Karima BENHAMZA

Benhamza.Karima@Univ-guelma.dz

Année universitaire : 2021 /2022

Avant-propos

L'importance de la logique formelle, de la théorie de la démonstration et de la théorie des modèle, autant pour les mathématiciens que pour les informaticiens, ne fait pas l'objet d'un avis unique. Beaucoup de chercheurs affirment qu'elles sont essentielles pour le fonctionnement de la recherche dans ces domaines.

Ce polycopié, destiné aux étudiants en formation de Master Systèmes Informatiques, présente le contenu du cours du module " **Logique et fondements de l'informatique 1**" de la première année conformément aux programmes d'enseignement du canevas proposé.

Le module de "Logique et fondements de l'informatique 1" se focalise sur ces trois axes centraux : la théorie de démonstration et les modèles de calculs et la calculabilité . Tous ces axes sont reliés par un objectif commun "Retrouver les capacités et les limites des ordinateurs".

Savoir-faire acquis en fin de cours

Ce polycopié vise à présenter les concepts de base et offrir un certain nombre de théories et de modèles permettant de représenter puis aider à résoudre différents problèmes concrets . Ainsi, il s'agit de développer les capacités des étudiants à concevoir et mettre en œuvre des modèles pertinents face à une situation de démonstration logique et de connaitre aussi la décidabilité et la calculabilité des problèmes.

A l'issue de ce module, l'étudiant maîtrisera les méthodes de démonstration et les modèles de calculabilité qui lui permettront de les utiliser de manière efficace dans le cadre de la résolution de problèmes concrets.

Pré-requis : Logique Mathématique

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

Faculté de Mathématiques,
D'Informatique et de Sciences de la Matière
Département : Informatique

جامعة 8 ماي 1945 قالمة
كلية الرياضيات، الإعلام الآلي و علوم المادة
قسم: الإعلام الآلي

Syllabus- Logique et fondement de l'informatique 1

Enseignant responsable du module : *Dr Benhamza Karima*

Grade *: Maître de Conférence A*

Email : Benhamza.karima@univ-guelma.dz

Bureau : *Département d'Informatique E8.2 bis*

Intitulé du Master : *Systèmes Informatiques* ***Intitulé de l'UE :*** *UEF1* ***Semestre :*** *01*

Intitulé de la matière : Logique et fondement de l'informatique 1

Volume horaire : Cours: *3h* ***TD****:1h30* ***TP*** *: 1h30*

Crédits : 7 **Coefficients : 4**

Objectifs de l'enseignement : Le but de ce module est de :

1- Introduire les étudiants à la logique mathématique et, en particulier à la théorie de la démonstration.
2- Fournir aux étudiants les bases nécessaires afin de pouvoir comprendre le fonctionnement de la plupart des outils de démonstration automatique développés en particulier dans le monde académique, et éventuellement de coder eux-mêmes un tel outil.

Connaissances préalables recommandées : Introduction à la logique mathématique

Contenu de la matière : Logique et fondements de l'informatique 1

1 Chapitre : Introduction

1.1 Récursivité et induction
1.2 Raisonnement par récurrence sur N
1.3 Formalisation : Premier théorème du point fixe
1.4 Applications
1.4.1 Quelques exemples
1.4.2 Arbres binaires étiquetés

2 Chapitre : Démonstrations

2.1 Introduction - Rappel Logique
2.2 Démonstrations à la Frege et Hilbert
2.3 Démonstration par déduction naturelle

2.3.1 Règles de la déduction naturelle

2.3.2 Validité et complétude

2.4 Démonstrations par résolution

3 Chapitre : Modèles de calculs

3.1 Machines de Turing

3.1.1 Ingrédients

3.1.2 Description

3.1.3 Programmer avec des machines de Turing

3.1.4 Techniques de programmation

3.1.5 Applications

3.1.6 Variantes de la notion de machine de Turing

3.1.7 Localité de la notion de calcul

3.2 Machines de Turing non-déterministes

3.3 Modèles rudimentaires

3.3.1 Machines à $k \geq 2$ piles

3.3.2 Machines à compteurs

3.4 Thèse de Church-Turing

3.4.1 Équivalence de tous les modèles considérés

4 Chapitre : Calculabilité

4.1 Machines universelles

4.1.1 Interpréteurs

Mode d'évaluation : Examen final + contrôle continu

Références

1. Bournez, Olivier. Fondements de l'informatique, Logique, modèles, et calculs, Cours INF423 de l'Ecole Polytechnique 2021.
2. Dehornoy Patrick. La théorie des ensembles. Calvage et Mounet, 2017.
3. Dowek Gilles, Les démonstrations et les algorithmes : introduction à la logique et à la calculabilité, Editions de l'Ecole Polytechnique, 2010.
4. Wolper, Pierre, Introduction à la calculabilité : cours et exercices corrigés. Dunod, 2001.
5. Stern, Jacques. Fondements mathématiques de l'informatique. Ed.Science International, Paris, 1994.
6. Lassaigne, Richard, Michel de Rougemont. Logique et fondements de l'informatique Hermes1993.

Sommaire

Chapitre 3 . Modèles de calculs

Chapitre 4. Calculabilité

Listes des figures

Introduction Générale

Introduction Générale

Le **raisonnement** est un processus cognitif permettant de poser un problème de manière réfléchie en vue d'obtenir un ou plusieurs résultats. L'objectif d'un raisonnement est de mieux comprendre les faits et/ou d'en vérifier d'autres, en faisant appel alternativement à différentes lois et ceci quel que soit le domaine d'application : mathématiques, informatique,...

Dans le domaine des mathématiques, le **raisonnement par récurrence** (appelé aussi raisonnement par induction) est une forme de raisonnement visant à démontrer une propriété portant sur les entiers naturels. Le raisonnement par récurrence établit une propriété importante: celle d'être construits à partir d'un entier n_0 en itérant le passage au successeur (de n vers n+1). Ce type de démonstration est très important lorsqu'on veut démontrer qu'une propriété, dépendant de n, est vraie pour toutes les valeurs de n.

D'autre part, ARISTOTE fut un des premiers à essayer de formaliser le raisonnement en utilisant la **logique**. Cette dernière permet de préciser ce qu'est un raisonnement correct, indépendamment du domaine d'application. Dans ce cas, le raisonnement est un moyen d'obtenir une conclusion à partir d'hypothèses données. En fait, un raisonnement logique correct ne dit rien sur la vérité des hypothèses, il dit seulement qu'à partir de la vérité des hypothèses, nous pouvons déduire la vérité de la conclusion par les lois de la logique propositionnelle ou de prédicats.

Dans la logique mathématique la notion de **"Système Formel"** est utilisé pour fournir une définition stricte du concept de démonstration (qui sera présenter par la suite). Un système formel est un ensemble de formules que l'on peut interpréter comme des noms, des phrases et encore de propositions. Il utilise des symboles primitifs (un alphabet) pour construire de manière définitive un langage formel à partir d'un ensemble d'axiomes à travers des règles de formation . Il se compose donc de formules valides construites par des combinaisons finies des symboles primitifs - combinaisons qui sont formées à partir des axiomes conformément aux règles énoncées.

Les trois problèmes fondamentaux pour les théories des systèmes formels est premièrement de définir les ensembles de formules, ensuite de les interpréter et enfin d'expliquer comment prouver les vérités des ensembles de formules .

Ainsi, et à partir du concept "raisonnement", un autre axe de recherche a émerger dans le fondement de l'informatique et qui est la " Théorie de la **Démonstration**" ou plus précisément " Théorie de la Preuve". Chaque étape d'une démonstration (preuve) est soit un axiome (fait acquis), soit l'application d'une règle qui permet d'affirmer qu'une proposition, la conclusion, est une conséquence logique d'une ou plusieurs autres propositions, les prémisses de la règle. Une proposition qui est la conclusion de l'étape ultime d'une démonstration est un théorème. Le système formel est ainsi utilisé pour déduire des théorèmes à partir d'axiomes selon un ensemble de règles. Ces règles, qui sont utilisées pour réaliser l'inférence de théorèmes à partir d'axiomes, sont le calcul logique du système formel.

La théorie de la **calculabilité** est un axe de recherche de l'informatique théorique. La calculabilité cherche à classifier les fonctions qui peuvent être calculées à l'aide d'un algorithme . Ainsi, une bonne appréhension de ce qui est calculable de ce qui ne l'est pas permet de voir les limites des problèmes que peuvent résoudre les ordinateurs. La thèse de Church postule que tout problème de calcul fondé sur une procédure algorithmique peut être résolu par une machine de Turing.

Dans ce polycopie, destiné aux étudiants en formation de Master 1 "Systèmes Informatiques", nous allons tenter de cerner le domaine de la Logique et les fondements de l'informatique en présentant les concepts, les définitions et les théorèmes en relation. Nous mettons l'accent principalement sur les théories de démonstration et de la calculabilité.

Ce manuel est structuré comme suit:

Chapitre 1: **Introduction**

Dans ce chapitre, on présente les concepts clés de la récursivité et de l'induction. Ces derniers vont aider à expliquer le raisonnement par récurrence à travers une multitude d'exemples. Enfin, le théorème du point fixe est formalisé pour l'ensemble des fonctions inductives. Des exemples applicatifs sur les arbres binaires étiquetés clôturent ce chapitre.

Chapitre 2: **Démonstration**

Ce chapitre présente, dans la partie introduction, les éléments de base de la logique ; le calcul propositionnel et le calcul des prédicats du premier ordre. dans les sections qui suivent, les concepts liés à la théorie de preuve ou démonstration vont être exposés. Les principaux systèmes formels de démonstration tels la Preuve à la Frege et Hilbert, la preuve par Déduction Naturelle (DN) et la Preuve par résolution vont être aussi détaillées avec des exemples d'applications.

Chapitre 3: **Modèles de calculs**

Dans ce chapitre, le principe de fonctionnement des machines de Turing est présenté suivi par une présentation des modèles rudimentaires. Finalement la thèse de Church-Turing est exposée.

Chapitre 4: **Calculabilité**

Dans ce dernier chapitre, la machine de Turing universelle est présentée pour appuyer les principes de "calculabilité". Son schéma fonctionnel est exposé. Le concept d'interpréteur est aussi abordé. Ce chapitre est clôturé avec des exemples de problèmes indécidables.

Finalement, des **séries d'exercices corrigés pour** chaque chapitre, un **énoncé de TP final avec son corrigé type et** une proposition d'**examen avec son corrigé type** ont été joints pour une meilleure maitrise des concepts développés dans ce module. Une conclusion générale et les sources bibliographiques clôturent ce polycopié.

Chapitre 1

Introduction

Chapitre 1

Introduction

Contenu du chapitre 1

Introduction

Le raisonnement est un "processus cognitif" qui permet d'obtenir de nouveaux résultats ou bien de vérifier la réalité d'un fait en faisant appel soit à différentes « lois » ou soit à des expériences. On dit que l'individu effectue des inférences et que le mécanisme d'élaboration de ces inférences s'appelle raisonnement.

En logique, le raisonnement déductif est défini comme une inférence dont la conclusion est aussi certaine que les prémisses. L'induction logique consiste à former des représentations générales à partir de faits particuliers (ou chercher des lois générales à partir de l'observation de faits particuliers).

Ce chapitre a pour objectif de formaliser les principes de l'induction – déduction et de rappeler les notions fondamentales et utiles à la logique. Ces notions seront par la suite des bases indispensables pour maitriser les différents modèles de raisonnement rencontrés dans ce module. Les notions et principes évoqués dans ce chapitre, sont essentiellement repris des ouvrages [Dowek, 2008], [Arnold and Guessarian, 2005] et [Dehornoy, 2017] et du polycopié de cours INF421 [Bournez,2021].

1.1 Récursivité et Induction

La récursivité et l'induction sont deux notions très liées. En effet, les deux concepts sont employés de façon parfois interchangeable, avec une frontière assez floue. On présente ci-dessous deux définitions de ces deux concepts:

1.1.1 Définitions

Définition 1: Formellement, l'induction est une technique de preuve mathématique employée pour démontrer des théorèmes sur des structures infinies munies d'un ordre bien fondé , souvent, des structures définies **récursivement** (nombres naturels).

Définition 2: La récursivité est une technique de définition et construction d'objets mathématiques (tels les fonctions, ensembles, etc.).

Dans les deux concepts, on étudie les propriétés d'un objet (les propriétés d'un entier n) en les réduisant aux propriétés d'objets plus petits (les propriétés de n−1). **Formellement, l'induction est une technique de preuve mathématique employée pour démontrer des théorèmes sur des structures infinies munies d'un ordre bien fondé .**

1.1.2 Notions de Base

a. Ensembles : Un ensemble est une collection d'éléments que l'on peut énumérer ou définir par une propriété.

Lorsqu'on énumère les éléments d'un ensemble, on dit que cet ensemble est défini par extension, lorsqu'on définit un ensemble par une propriété, on dit que cet ensemble est défini par compréhension.

Un ensemble qui ne contient aucun élément s'appelle : l'ensemble vide noté ∅.

Exemple : N ensemble des entiers naturels, N = {1, 2, 3, ..}.

b. Propriétés

- Soit E un ensemble, et e un élément. On note e∈E pour signifier que e est un élément de l'ensemble E.

- Si A et B sont deux ensembles, on note A⊂B pour signifier que tout élément de A est un élément de B. On dit dans ce cas que A est une partie de B.

- Lorsque E est un ensemble, les parties de E constituent un ensemble que l'on note P(E). On notera A∪B, A∩B pour respectivement l'union et l'intersection des ensembles A et B. Lorsque A est une partie de E, on notera A^c pour le complémentaire de A dans E.

c. Fonctions : On appelle produit cartésien, de deux ensembles E et F, noté ExF, l'ensemble des couples formés d'un élément de E et d'un élément de F :

$$\mathbf{ExF = \{(x,y) | x \in E \text{ et } y \in F\}.}$$

Une application f d'un ensemble E vers un ensemble F est un objet qui associe à chaque élément e d'un ensemble E un unique élément f(e) de F.

Formellement, une fonction f d'un ensemble E vers un ensemble F est une partie Γ de ExF tel que pour tout x ∈ E il y a au plus un y ∈ F avec (x,y) ∈Γ.

d. Graphe :

1- Définition: Un graphe G=(V,E) est donne par un ensemble V, dont les élément sont appelés **sommets** (Vertex) et d'une partie E ⊂ VxV dont les éléments sont appelés les **arcs (Edges).**

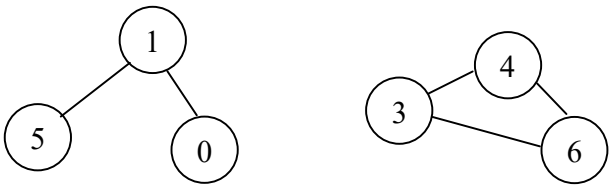


Figure 1.1. Exemples de graphe

Exemple :

Soit un graphe G = (V,E) tel que:

V = {0.1.2.3.4.5.6}, E = {(0,1), (3,4), (5,1), (6,3), (6,4)}.

2- Propriétés d'un graphe: On présente ci-dessous quelques propriétés :

Soit e=(u,v) :

- u et v sont dit voisin s'il y a une arête entre u et v .
- Le degré de u est le nombre de voisin de u.
- Un chemin entre deux sommets 's' et 't' est la suite de sommets entre 's' et 't'
- Le sommet 's' est à distance n de 't' s'il existe un chemin de longueur n entre les sommets 's' et 't'.

e. Arbre : Un graphe connexe sans cycle est appelé un arbre. On distingue souvent un sommet que l'on appelle sa racine. Pour tout sommet u, il existe un unique chemin (simple) entre la racine r et u.

Tout sommet a sur ce chemin est appelé un ancêtre de u et u est dit un descendant de a. L'avant dernier sommet p sur ce chemin est appelé le père de u et u est appelé un fils de p. Le sous - arbre de racine u est l'arbre induit par les descendants de u.

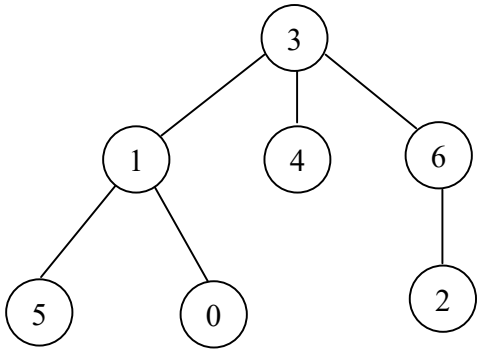


Figure 1.2. Exemple d'un arbre

- L'arité d'un sommet est le nombre de ses fils.
- Un sommet qui n'a pas de fils est appelé une feuille. Exemple : 0, 2, 4 et 5.

1.1.3 Alphabet, Mot, Langage

a. Un **alphabet** Σ est un ensemble fini, dont les éléments sont appelés des lettres.

b. Un **mot** w sur Σ est une suite finie d'éléments de Σ . w s'écrit $w = a_1a_2 \ldots a_n$, avec $a_i \in \Sigma$. $n \geq 0$ est la longueur du mot. Le mot vide, noté ε , est (l'unique) mot de longueur 0.

c. **Opérations sur les mots** : L'ensemble des mots Σ sur est noté Σ^*.
Etant donnés deux mots m_1 et m_2, la **concaténation** de m_1 et de m_2, notée $m_1.m_2$, ou notée m_1m_2, est le mot obtenu en mettant m_2 à la fin de m_1. La **concaténation** est une opération associative, avec un élément neutre, le mot vide.

Exemple : aba.ba est le mot ababa sur $\Sigma = \{a,b\}$

d. Un **langage** sur Σ est un ensemble de mots sur Σ (une partie de Σ^*).
Exemples de Langages : $\Sigma_{bin} = \{0,1\}$: $101101 \in \Sigma^*_{bin}$;

$\Sigma_{nombre} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$: $1794 \in \Sigma^*_{nombre}$, $007 \in \Sigma^*_{nombre}$

$\Sigma exp = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9,+,-,*,/,(,)\}$: alphabet des expressions arithmétiques. Arith est un langage des expressions arithmétiques : $((1+3)*2) \in$ Arith

Soit P un ensemble de variables. $\Sigma prop = P \cup \{0,1, \neg, \vee, \wedge, \Rightarrow, \Leftrightarrow,(,)\}$ est l'alphabet du calcul propositionnel sur P: $P=\{p,q,r\}$ $((r \vee p) \wedge q) \in \Sigma^* prop$.

1.2. Raisonnement par récurrence sur N

Les définitions récursives sont omniprésentes en informatique. Elles sont présentes dans de nombreux concepts manipulés. L'**induction** est un outil formel, simple et très utile en informatique qui permet de faire des **définitions récursives** ; et qui offre une technique de preuve qui **généralise la preuve par récurrence.**

Lorsque l'on raisonne sur les entiers, le premier principe d'induction aussi appelé principe de **récurrence mathématique sur N** est un mode de raisonnement très utile.

Exemple : L'ensemble des entiers naturel est le plus petit ensemble qui contient 0 et tel que si n est un entier naturel, alors n +1 est un entier naturel.

1.2.1 Définition inductive : Une définition inductive d'une partie X d'un ensemble consiste à donner :

- une donnée explicite de certains éléments de X (Base B) ;
- une donnée de règles de construction de nouveaux éléments de X à partir d'éléments déjà construits (étapes Inductives I).

Exemple : Une définition inductive des entiers pairs :

(B) $0 \in P$;

(I) $n \in P \Rightarrow n+2 \in P$

1.2.2 Principe général d'une définition inductive: Une partie X se définit inductivement si on peut la définir avec la donnée explicite de certains éléments de X et de moyens de construire de nouveaux éléments de X à partir d'éléments de X. Dans une définition inductive:

- certains éléments de l'ensemble X sont donnés explicitement;
- les autres éléments de l'ensemble X sont définis en fonction d'éléments appartenant déjà à l'ensemble X selon certaines règles.

1.3 Formalisation Théorème du point fixe

Théorème du Point Fixe. *A une définition* ***inductive*** *correspond un plus petit ensemble X qui vérifie les propriétés suivantes :*

(B) *il contient B : B* $\subset X$ *;*

(I) *il est stable par les règles de R : pour chaque règle* $r_i \in R$*, pour tout* $x_1 \ldots\ldots x_{ni} \in X$*, on a* $r_i(x_1 \ldots\ldots x_{ni}) \in X$.

On dit que cet ensemble est alors défini ***inductivement.***

1.3.1 Définition inductive de fonctions :

Théorème . *Soit* $X \subseteq E$ *un ensemble défini à partir de B et R de façon non-ambiguë. Soit Y un ensemble. Pour qu'une application f de X dans Y soit parfaitement définie, il suffit de se donner :*

(B) *la valeur de f(x) pour chacun des éléments* $x \in B$*;*

(I) *pour chaque règle* $r_i \in R$*, la valeur de f(x) pour* $x = r_i(x_1 \ldots\ldots x_{ni})$ *en fonction de la valeur de* $x_1 \ldots\ldots x_{ni}$, $f(x_1)$, . . ., *et* $f(x_{ni})$

Exemple : La fonction factorielle Fact de N dans N se définit **inductivement** par :

(B) Fact(0)= 1;

(I) Fact(n+1)=(n+1)* Fact(n).

1.3.2 Règle de déduction : On note souvent une définition inductive avec une autre notation qui utilise les règles de déduction :

$$\frac{}{B} \qquad \frac{x_1(x_1 \ldots\ldots x_{ni}) \in X}{r_i(x_1 \ldots\ldots x_{ni}) \in X}$$

La ligne horizontale désigne une règle de déduction, les hypothèses sont représentées au-dessus et les conclusions en dessous. Si ce qui est en dessus est vide alors la conclusion est valide sans hypothèses.

Exemple : L'ensemble N défini par :

$$\frac{}{0 \in N} \qquad \frac{n \in N}{n+1 \in N}$$

1.3.3 Arbre de dérivation : La propriété d'induction-déduction permet de garder la trace sur chaque élément obtenu en partant de X et en appliquant les règles.

Exemple : 1+2+3 ∈ Arith :

$$\dfrac{\dfrac{\dfrac{}{1 \in Arith} \quad \dfrac{}{2 \in Arith}}{1+2 \in Arith} \quad \dfrac{}{3 \in Arith}}{1+2+3 \in Arith}$$

Arbre de dérivation

On dit qu'une définition inductive est non ambiguë s il n'existe un qu'unique arbre de dérivation pour chaque élément de X. La définition précédente de Arith est ambiguë : Le mot 1+2+3 correspond aux dérivations suivantes :

$$\dfrac{\dfrac{\dfrac{}{1 \in Arith} \quad \dfrac{}{2 \in Arith}}{1+2 \in Arith} \quad \dfrac{}{3 \in Arith}}{1+2+3 \in Arith} \qquad \dfrac{\dfrac{}{1 \in Arith} \quad \dfrac{\dfrac{}{2 \in Arith} \quad \dfrac{}{3 \in Arith}}{2+3 \in Arith}}{1+2+3 \in Arith}$$

Arbre de dérivation 1 **Arbre de Dérivation** 2

1.4 Applications

Les applications de ces principes sont innombrables en informatique :

- Définition de structure de données récursives (arbres, chaines de caractère, listes, etc.);
- Calcul de fonctions et vérification de propriétés sur ces structures ;
- Définition de la syntaxe et de la sémantique de langages de programmation ;
- Fonctions récursives sur N : dénombrement d'ensembles et analyse de complexité.

1.4.1 Quelques exemples

Exemple 1 : Suite de Fibonacci

La suite de Fibonacci est un exemple de définition récursive basée non seulement sur la valeur de la fonction à n−1, mais aussi sur sa valeur à n−2. Par conséquent, deux cas de base sont nécessaires afin d'éviter une définition mal formée:

(B) F(0)=0,

(B) F(1)=1,

(I) F(n)=F(n−1)+F(n−2) pour n>1.

Exemple 2 : Le langage L des expressions entièrement parenthésées formées à partir d'identificateurs pris dans un ensemble A et des opérateurs + et x correspond à la partie de E = (A ∪ {+,x} ∪ {(,)}* qui est définie inductivement par:

(B) A ⊂ L;

(I) e, f ∈ L ⇒ (e + f) ∈ L;

(I) e, f ∈ L ⇒ (e x f) ∈ L

Exemple 3: Palindrômes Une chaine de caractères est palindrome si elle se lit de la même façon de gauche à droite et de droite à gauche. Par exemple, *abba* et *radar* sont palindromes. On peut définir la "palindromie" de la façon suivante:

(B) La chaîne vide est palindrome,

(B) Toute chaîne d'un seul caractère est palindrome,

(I) Si S est un palindrome, alors xSx est palindrome pour tout caractère x.

1.4.2 Solution algorithmique récursive

Disposer d'une solution algorithmique récursive à un problème permet d'écrire un programme plus simple résolvant ce problème. La découverte de telles solutions est parfois complexe mais rentable en terme de simplicité d'expression des programmes.

Exemple 4 : Algorithme récursif de calcul du PGCD (Plus Grand Diviseur Commun) de deux nombres non nuls :

```
Fonction PGCD(a, b: entier): entier;
  Var    résultat: entier;
  Debut
      Si(a = b) Alors
      résultat := a;
      Sinon
          Si (a > b) Alors
                  résultat := PGCD(a - b, b);
          Sinon
                  résultat := PGCD(a, b - a);
      FinSi
              FinSi
              Renvoyer résultat;
  Fin
```

La programmation récursive, pour traiter certains problèmes, est très économique pour le programmeur ; elle permet de les résoudre avec très peu d'instructions. En revanche, elle est très coûteuse en ressources car à l'exécution, la machine va être obligée de créer autant de variables temporaires que de fonction en attente.

Exemple 5: Ecrire une solution récursive pour le tri par fusion d'un tableau T(N)

```
Procedure Tri_Fusion (Var t : TAB; g, d : entier);
Var m, i, j, k : entier;  s : TAB;  // Type TAB = Tableau de N Reel ;
Debut
    Si d > g Alors
     Debut
        m ←(g + d) Div 2;
        Tri_Fusion (t, g, m);
        Tri_Fusion (t, m + 1, d);
         Pour i←m a g Faire
             s(i) ← t (i);
         FinPour
        Pour j ← m + 1 a d Faire
              s (d + m + 1 – j) ← t(j);
        FinPour
         i ← g;
         j ← d;

        Pour k ← g a d Faire
           Debut
              Si s(i) < s(j) Alors
              Debut
                   t(k) ← s (i);
                   i ← i + 1;
               Fin
              Sinon
               Debut
                   t(k) ← s(j);
                   j ← j - 1;
              Fin;
                 FinSi
           Fin
        FinPour
     Fin
Fin;
```

1.4.3 Arbres binaires étiquetés

Dans sa définition la plus générale, un arbre est un graphe connexe acyclique enraciné. On s'intéresse particulièrement aux arbres binaires, c'est à dire ceux dont chaque nœud a au plus deux fils. Un arbre binaire est donc soit un arbre vide, soit formé de deux arbres binaires, appelés fils gauche et fils droit.

a)- Définition d'un arbre binaire: Un arbre binaire sur un ensemble fini de sommets est soit vide, soit l'union disjointe d'un sommet appelé sa racine, d'un arbre binaire appelé sous-arbre gauche(A_g) et d'un arbre binaire appelé sous-arbre droit (A_d).

Un arbre binaire est donc représenté par un arbre binaire non vide sous la forme d'un triplet $A=(A_g,r,A_d)$.

b)-Définition inductive d'un arbre binaire : L'ensemble AB des arbres binaires étiquetés par l'ensemble A est la partie de Σ^*, où Σ est l'alphabet $\Sigma=A\cup\{\varnothing,(,),,\}$ définie inductivement par :

- (B) $\varnothing \in AB$;
- $g, d \in AB \Rightarrow (g, a, d) \in AB$, pour chaque $a \in A$.

c)-Définition d'un arbre binaire étiqueté : Un arbre binaire étiqueté, est soit l'arbre vide, soit formé d'une étiquette (de n'importe quel type: entier,..) et de deux arbres, appelés fils gauche et fils droit. Un sous-arbre d'un arbre binaire est soit un fils, soit un sous-arbre d'un fils. Un arbre vide n'a pas de sous-arbre. Les *feuilles* d'un arbre sont des sous-arbres dont les deux fils sont des arbres vides.

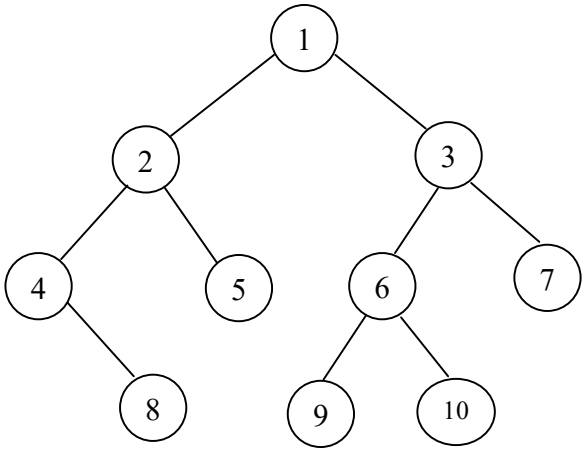


Figure 1.3. Arbre binaire étiqueté (par des entiers)

Exemple A. Expressions arithmétiques.

On peut représenter les expressions arithmétiques par des arbres étiquetés par des opérateurs, des constantes et des variables. La structure de l'arbre rend compte de la priorité des opérateurs et rend inutile les parenthèses.

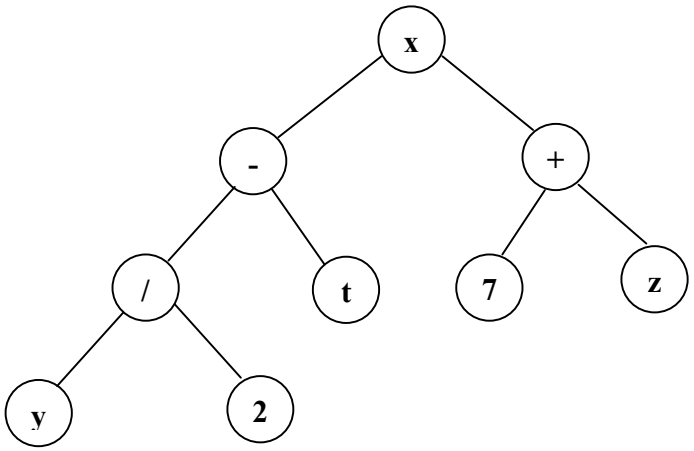


Figure 1.4. Arbre binaire étiqueté d'une expression arithmétique

Exemple B. Arbre de Fibonacci

Un arbre de Fibonacci d'ordre p est :

– l'arbre vide si p = 0 ;

– un arbre réduit à une feuille si p = 1 ;

– un arbre dont le fils gauche est un arbre de Fibonacci d'ordre p − 1 et le fils droit un arbre de Fibonacci d'ordre p − 2 si p > 2.

Squelette d'un arbre de Fibonacci

Les squelettes des arbres de Fibonacci d'ordre 2, 3, 4 et 5 sont dessinés ci-dessous :

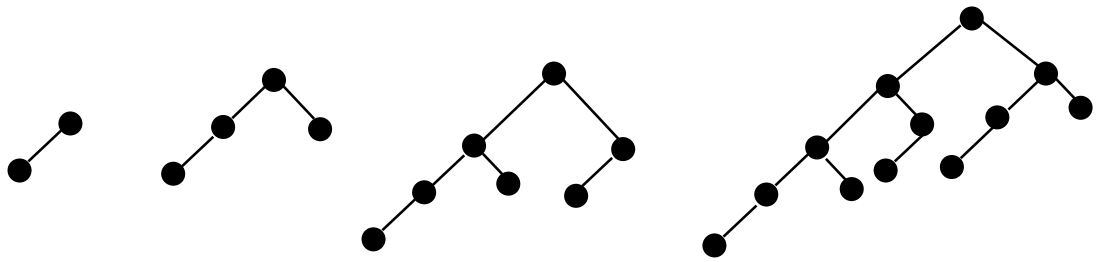


Figure 1.5. Arbres binaires de Fibonacci

Si fp désigne le nombre de feuilles d'un arbre de Fibonacci d'ordre p alors :

- $f_0 = 0$
- $f_1 = 1$;
- $\forall p \geq 2$; $f_p = f_{p-1} + f_{p-2}$

donc le nombre de feuilles f_p d'un arbre de Fibonacci d'ordre p est égal au $p^{ième}$ nombre de Fibonacci.

e. Parcours d'un arbre binaire. Le parcours le plus utilisé dans les arbres binaire est le parcours dit en profondeur d'abord.

Son principe est le suivant : pour parcourir un arbre non vide , on parcourt récursivement son sous-arbre gauche, puis son sous-arbre droit, la racine de l'arbre pouvant être traitée au début (ordre préfixe), entre les deux parcours (ordre infixe) ou à la fin (ordre postfixe).

d**. Arbre Binaire de Recherche** : Un arbre binaire de recherche (ou ABR) est une structure de donnée qui permet de représenter un ensemble de valeurs si l'on dispose d'une relation d'ordre sur ces valeurs. C'est donc arbre binaire étiqueté par les éléments tel que pour tout nœud p d'étiquette x:

- toutes les étiquettes de l'enfant gauche de p sont inférieures à x
- toutes les étiquettes de l'enfant droit de de p sont supérieures à x

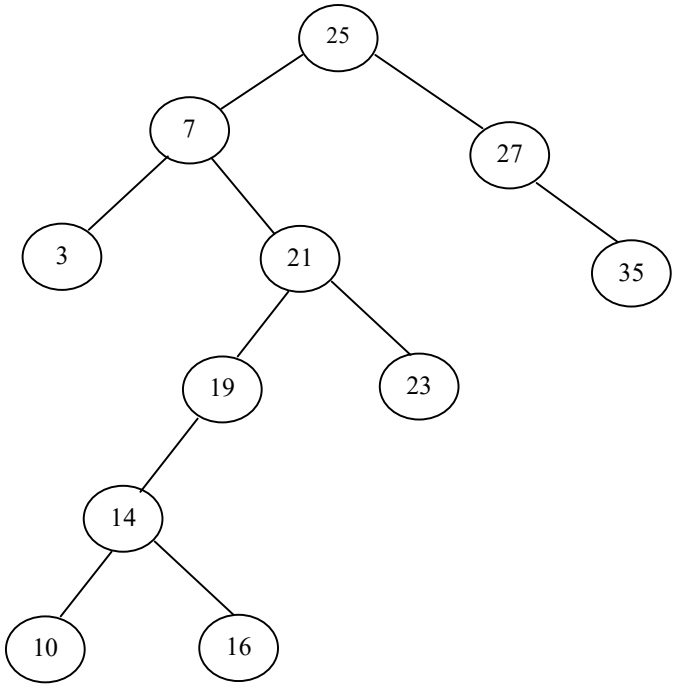


Figure 1.6. Exemple d'arbre binaire de recherche

Les opérations caractéristiques sur les arbres binaires de recherche (ABR) sont l'insertion, la suppression, et la recherche d'une valeur. Cette dernière consiste à parcourir une branche en partant de la racine, en descendant chaque fois sur le fils gauche ou sur le fils droit suivant que la valeur portée par le nœud est plus grande ou plus petite que la valeur cherchée selon l'algorithme ci-dessous.

```
Fonction recherche(a, v): Booléen
entrée : a est un ABR, v est une valeur.
sortie : Vrai si v figure dans a et Faux sinon.
début
    si est vide(a) alors
        retourner Faux
    sinon
        si v == val(a) alors
        retourner Vrai
        sinon
            si v < val(a) alors
            retourner recherche(v,fils gauche(a))
            sinon
            retourner recherche(v,fils droit(a))
            finsi
        finsi
    finsi
fin
```

Pour l'insertion d'une nouvelle valeur, le principe est le même que pour la recherche. Un nouveau nœud est créé avec la nouvelle valeur et inséré à l'endroit où la recherche s'est arrêtée.

L'algorithme de l'insertion est le suivant:

```
Fonction Insertion: Insertion d'une nouvelle valeur dans un ABR
entrée : a est un ABR, v est une valeur.
résultat : v est insérée dans a
début
    si est vide(a) alors
        a=cree_arbre(v,cree_arbre vide(),cree_arbre vide())
    sinon
    si v < val(a) alors
        Insertion(v,fils gauche(a))
    sinon
    si v > val(a) alors
        Insertion(v,fils droit(a))
    finsi
    finsi
    finsi
fin
```

La suppression dans un arbre binaire de recherche consiste d'abord à recherche le nœud à supprimer. S'il a deux fils, on le remplace par l'extrémité du bord gauche du sous arbre droit, et on supprime cette extrémité. Le nœud à supprimer, que ce soit le nœud initial ou l'extrémité du bord droit a ainsi au plus un fils. Il suffit alors de remonter le fils restant à la place du nœud à supprimer.

L'algorithme de suppression est le suivant:

```
Fonction Supprimer ( Arbre Elt ) : Arbre
entrée : a est un ABR, v est une valeur.
résultat : v est supprimer dans a
Node : Noeud
Début
    Si Arbre = ∅ alors     Afficher Elt non trouvé dans l'arbre;
    sinon
     Si clef ( Elt )  < clef ( Arbre.Racine )   alors
              Supprimer ( Arbre.filsG Elt ) //on cherche à gauche
    Sinon
    Si clef ( Elt )  >  clef ( Arbre.Racine ) alors
                      Supprimer ( Arbre.filsD Elt ) //on cherche à droite
    Sinon //l'élément est dans ce noeud
    Si Arbre.filsG = ∅ alors //sous-arbre gauche vide
              Arbre ←Arbre.filsD//remplacer arbre par son sous-arbre droit
    Sinon
          Si Arbre.filsD = ∅ alors //sous-arbre droit vide
            Arbre ←Arbre.filsG //remplacer arbre par son sous-arbre gauche
             Sinon//le noeud a deux descendants
               Node ←PlusGrand( Arbre.filsG ); //Node = le max du fils gauche
               clef ( Arbre.Racine ) ←clef ( Node );//remplacer valeur (étiquette)
               détruire ( Node ) //on élimine ce noeud
                     Fsi
        Fsi
        Fsi
        Fsi
       Fsi

 Fin
```

Remarque:

1. La manipulation des arbres binaires permet de mettre en évidence l'importance de la définition inductive.

2. L'équilibre d'un arbre binaire est un entier qui vaut 0 si l'arbre est vide et la différence des hauteurs des sous-arbres gauche et droit de l'arbre sinon. Un arbre binaire est équilibré lorsque l'équilibre de chacun de ses sous-arbres non vides n'excède pas 1 en valeur absolue.

Conclusion

Une définition récursive explique comment construire un élément d'un type donné à partir d'élément du même type plus simple. Dans ce chapitre, on a présenté les concepts clés de la récursivité et de l'induction. Ces derniers ont permis d'expliquer le raisonnement par récurrence à travers de multiple exemples et de formaliser le théorème du point fixe pour l'ensemble des fonctions inductives. Les notions et principes d'arbre binaire, utiles à la suite du cours a été finalement abordé. Dans le second chapitre, on présentera la théorie de preuve ou de démonstration. Cette dernière est basée sur des méthodes fondamentales pour démontrer la validité du raisonnement.

Chapitre 2

Démonstration

Chapitre 2
Démonstration

Contenu du chapitre 2

2.1 Introduction

Il existe dans la théorie de la preuve, différentes méthodes qui permettent de démontrer la validité – ou la satisfiabilité- d'une formule ou encore déduire une formule à partir d'un certain nombre de formules (hypothèses).

Même si l'approche sémantique permet de décider si un énoncé est une tautologie, une contradiction ou ni l'une ni l'autre. Elle ne peut pas être appliquée pour une bonne partie de la démonstration logique du fait de l'explosion combinatoire avec un nombre important nombre de variable propositionnelles.

En fait, une démonstration ne correspond pas à explorer tous les cas d'une façon systématique. En conséquence une autre approche, dite déductive, basée sur la formalisation de preuve et de la théorie de la démonstration a été proposée.

Ce chapitre a pour objectif de présenter cette approche avec ses méthodes et concepts et qui s'insèrent dans le contexte de la théorie de preuve . Dans l'introduction, on exposera un rappel de la logique propositionnelle suivi des notions clés de la logique de prédicat indispensables pour l'introduction de la théorie de la démonstration qui sera exposée dans la partie suivante.

Les principes et méthodes évoqués dans ce chapitre, sont essentiellement repris des ouvrages [Lassaigne, 93], [Stern, 94], [Arnold and Guessarian, 2005], [Dowek, 2008], [Dehornoy, 2017] et du polycopié de de cours INF421 [Bournez,2021].

2.1.1 Logique Propositionnelle

2.1.1.1 Introduction

La logique est une science relative aux processus de la pensée rationnelle (induction, déduction, hypothèse) et à la formulation discursive des vérités. La logique est exploitée dans les perspectives de la démonstration automatique suivantes:

a. **Calcul logique** : Mécanismes permettant d'automatiser la démonstration logique par des calculs symboliques et de mener l'inférence.

b. **Résolution de problèmes** : Modelisa1on en logique pour permettre sa résolution automatique.

La manière de traiter cette notion, les fondements, le formalisme utilisé, etc., changent d'une *logique* à une autre (Figure 2.1)

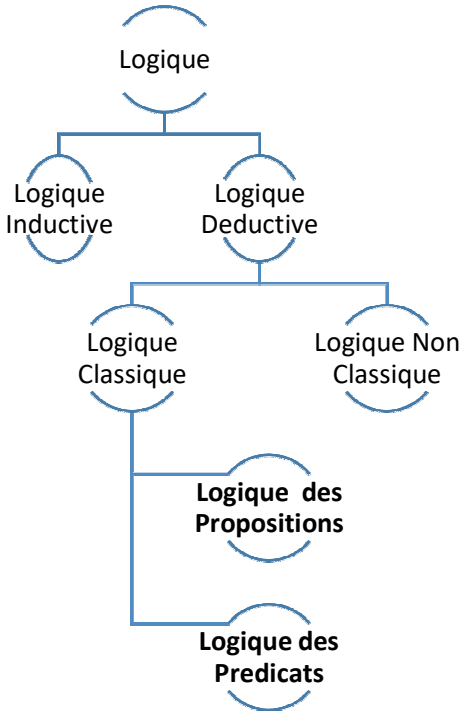


Figure 2.1. Différentes logiques

Toutefois, le besoin d'un langage formel est essentiel pour étudier le raisonnement. Ceci permettra de garantir que l'évaluation des raisonnements se fasse d'une manière automatique.

2.1.1.2 Concepts de bases

a) **Assertion (Proposition)**: Une assertion est un énoncé mathématique auquel on attribue l'une des deux valeurs logiques : le vrai (V) ou le faux (F).

Exemple: L'assertion 1 + 1 = 2 est vraie.

L'assertion 2 + 2 = 5 est fausse.

b) **Tautologies et antilogies :** Les assertions (dépendantes de P et Q) qui sont **vraies** quelle que soit la valeur de vérité de P et Q sont dites des **tautologies**.

Une tautologie est en fait un théorème de logique. Les assertions (dépendantes de P et Q) qui sont **fausses** quelle que soit la valeur de vérité de P et Q sont dites des **antilogies.**

c) **Un axiome** est un énoncé supposé vrai à priori et que l'on ne cherche pas à démontrer.

2.1.1.3 Formule propositionnelle

Les formules propositionnelles sont définies à l'aide de constantes, variables et connecteurs tels que :

- Les constantes sont V et F ;
- Les variables *propositionnelles* (ou *formules atomiques*, ou encore atomes) forment un ensemble dénombrable, désignées par des lettres romaines;
- Un connecteur unaire ¬,
- et Quatre connecteurs binaires : ∨, ∧, ⇒ et ⇔.

a. Définition inductive de Formule propositionnelle

Soit P un ensemble de variables. L'ensemble des formules propositionnelles F sur P est le langage sur l'alphabet P ∪ {¬,∨, ∧, ⇒ ,⇔} défini **inductivement** par les règles suivantes :

- **(B) T**oute variable propositionnelle est une formule propositionnelle ;
- **(I)** Si F ∈ F alors ¬F ∈ F ;
- **(I)** Si F,G ∈ F alors (F ^ G) ∈ F, (F ∨ G) ∈ F, (F ⇒ G) ∈ F, et (F ⇔ G) ∈ F.

L'ensemble des formules (ou propositions) de la logique propositionnelle est donc le plus petit ensemble de mots construits sur l'alphabet tel que :

- *V (Vrai) et F* (faux) sont des formules ;
- si A est une formule atomique alors A est une formule;
- ¬ A est une formule si A est une formule ;
- (A ∨ B), (A ∧ B), (A ⇒ B) et (A ⇔ B) sont des formules si A et B sont des formules.

Remarque : Si on n'utilise pas des parenthèses, l'ordre de priorité des connecteurs est comme suit : ¬ , ∧ , ∨, ⇒, ⇔ , et l'associativité est à gauche pour chaque connecteur.

b. Représentation de la structure d'une formule propositionnelle

La structure d'une formule est souvent mise en évidence par une représentation en arbre. où les nœuds représentent les connecteurs.

Exemples: **(a∨ (¬ b∧c))** **((p ∧ ¬ (p ∨ q)) ∧ ¬ r)**

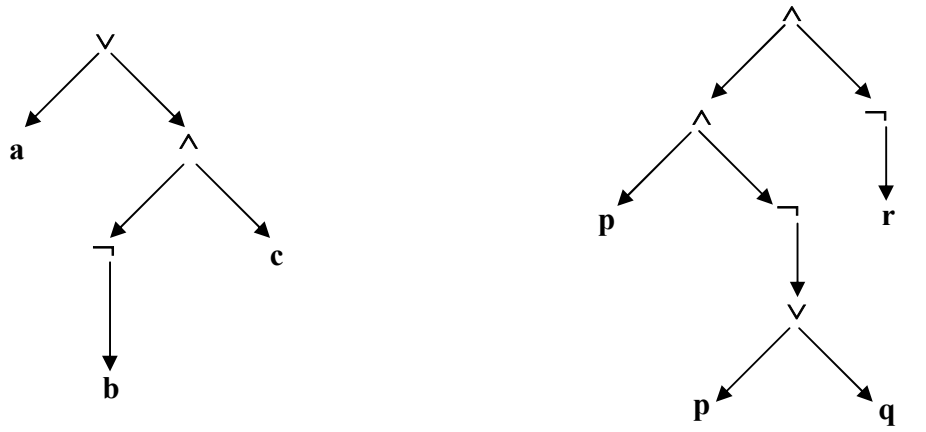


Figure 2.2. Exemples de représentations en arbre de formules

c. Sémantique d'une formule propositionnelle

La sémantique (valeur de vérité) d'une formule est une valuation ou fonction v de P vers {0,1} i.e. v : v: P→ {0,1}. Ainsi, toute fonction des valuations $f : \{0,1\}^n \rightarrow \{0,1\}$ est la valeur de vérité d'une formule propositionnelle.

p	q	¬p	p ∨ q	p ∧ q	p ⇒ q	p ⇔ q
0	0	1	0	0	1	1
0	1	1	1	0	1	0
1	0	0	1	0	0	0
1	1	0	1	1	1	1

Figure 2.3. Table de vérité

Lorsque F s'évalue en 1, on dit que F est **satisfaite** en v, et que **v est un modèle de F**.

2.1.1.4 . Propriétés du Langage Propositionnel (LP)

Plusieurs propriétés découlent des définitions précédentes:

a. Interprétation : Une *interprétation* I (ou valuation) est une application de l'ensemble des variables propositionnelles dans l'ensemble des valeurs de vérité {V,F} (ou {0,1}).

Une interprétation donnée I peut être *étendue* à l'ensemble des formules comme suit (A et B étant des formules):

- *I(F) = F (ou 0) et I(V) = V (ou 1)*
- *I(¬A) = V si I(A)= F et I(¬A) = F sinon (ou 1 - I(A))*
- *I(A ∧ B) = V si I(A)= V et I(B) =V et I(A ∧ B) = F sinon (ou min(I(A),I(B)))*

- *$I(A \vee B) = V$ si $I(A) = V$ ou $I(B) = V$ et $I(A \wedge B) = F$ sinon (ou $\max(I(A), I(B))$)*
- *$I(A \Rightarrow B) = V$ si $I(A) = F$ (ou 0) ou $I(B) = V$ (ou 1) et $I(A \Rightarrow B) = F$ sinon.*
- *$I(A \Leftrightarrow B) = V$ si $I(A \Rightarrow B) = V$ (ou 1) et $I(B \Rightarrow A) = V$ (ou 1) et $I(A \Leftrightarrow B) = F$ sinon.*

b. Modèle : On dit que I est un **modèle** pour une formule A (ou I satisfait A) si et seulement si I(A)=V. Aussi I est un modèle pour un ensemble de formules S si et seulement si I est un modèle pour toute formule A de S.

c. Validité et Satisfiabilité : Soit A une formule, on dit que A est **valide** (ou tautologique ; noté |= A) si I(A) = V pour toute interprétation I. Sinon A est **invalide** ou falsifiable. A est **satisfaisable** si et seulement s'il existe une interprétation I tel que I(A) = V. Sinon A est non satisfaisable ou **contradictoire**.

d. Consistance : Soit S un ensemble de formules. S est ***inconsistant*** s'il n'existe aucun modèle pour S, autrement dit, **un modèle pour lequel toutes les formules de S ont simultanément la valeur vraie.** Si un tel modèle existe S est dit ***consistant*** ou satisfaisable.

e. Conséquence logique : Une formule A est conséquence logique de n formules A1, ... An, noté {A1, ... An} |= A, si et seulement si tout modèle de {A1, ... An} est un modèle de A.

f. Équivalence tautologique : Soit A1 et A2 deux formules, elles sont dites tautologiquement équivalentes si A1 |= A2 et A2 |= A1.

2.1.2 Formes Normales et Clausales

2.1.2.1 Définitions

Un **littéral** est une formule atomique ou la négation d'une formule atomique

(p et ¬ q avec p et q atomes).

Une formule est une **Forme Normale Disjonctive (**FND) si elle est une **disjonction** de **conjonctions** de littéraux. Une **clause** est une disjonction de n ($n \geq 0$) littéraux.

Une formule est **une Forme Normale Conjonctive (FNC)** si elle est une conjonction de clauses.

Exemple: $\neg p_1 \vee p_2 \vee p_3$.

2.1.2.2 Mise en Forme Normale Conjonctive et Disjonctive

Toute formule propositionnelle est équivalente à une forme normale conjonctive et à une forme normale disjonctive.

Pour mettre une formule en forme FNC ou FND on applique l'algorithme suivant :

Algorithme de mise en forme normale conjonctive
Entrée : une formule A1 **Sortie** : une FNC A'1 équivalente à A1
Début
Eliminer $\Leftrightarrow$ et $\Rightarrow$
Appliquer à A1 les remplacements suivants autant de fois que nécessaire :
$\neg \neg A \rightarrow A$
$\neg (A \vee B) \rightarrow \neg A \wedge \neg B$
$\neg (A \wedge B) \rightarrow \neg A \vee \neg B$
$A \vee (B \wedge C) \rightarrow (A \vee B) \wedge (A \vee C)$
Fin

Figure 2.4. Algorithme de mise en forme FNC ou FND

2.1.2.3 Validité d'une formule en forme Clausale

Si une formule est en forme normale clausale alors il est intéressant de retrouver le(s) modèle (s) ou valider la formule. Pour vérifier la validité d'une clause, on applique les énoncés suivants:

1. Une formule en FNC est valide si et seulement si toutes ses clauses sont valides.
2. Une clause est valide si et seulement si elle contient deux littéraux opposés, c-à-d, de la forme L_1 v ... v p v ... v ¬p v ... v L_n.
3. Une clause C = L_1 v ... v L_i v ... v L_n peut s'écrire comme étant l'ensemble de ces littéraux: C = {L_1 , ... , L_i , ... , L_n } .

Une forme normale conjonctive A= $C_1 \wedge \ldots \wedge C_i \wedge \ldots \wedge C_n$ est écrite en forme clausale sous la forme d'un ensemble de clauses A = {C_1 , ... , C_i , ... , C_n} ou d'une suite de clauses C_1 , ... , C_i , ... , C_n.

4. A est donc valide si et seulement si toutes les clauses qu'elle contient sont valides.

Remarque: *Pour montrer la validité (resp. satisfiabilité) d'un ensemble de clauses, on peut ignorer les clauses tautologiques (c-a-d Élimination des clauses valides).*

2.1.3 Logique des Prédicats

Le calcul des propositions est simple mais ne peut pas rendre compte de la totalité des raisonnements. Il ne permet pas faire allusion aux propriétés d'une variable. Il ne permet pas non plus de décrire des relations entre plusieurs variables et de généraliser le calcul des propositions afin de mieux formaliser le raisonnement.

A. Alphabet de la logique des prédicats

L'*alphabet* de la logique des prédicats LP1 est constitué de :

- Un ensemble dénombrable de constantes notées a,b,c,..
- un ensemble dénombrable VAR de **variables** d'objets ou d'individus, notées, x,y,z,...
- un ensemble dénombrable F de symboles de **fonctions** à n arguments (n≥0), notées, f, g, ... , âge-de, ...
- un ensemble dénombrable R de symboles de **prédicats ou relations** à n arguments (n≥0), notées ou , *P, Q, R*, …
- les *connecteurs :* ¬ , ∨ , ∧ , ⇒ et ⇔
- les quantificateurs ", $
- les séparateurs ‘(‘ et ‘)‘.

Exemples de propositions en logique des predicats

1. Si Jean est un être humain alors il est mortel

être-humain(Jean) → mortel(Jean)

2. Tout être humain est mortel

∀x (être-humain(x) → mortel(x))

3. Tous les chemins mènent à Rome

∀x (Chemin(x)→ mène_a (x. Rome))

2.3.2 Termes, Formule atomique

L'ensemble des **termes** de Langage de Prédicat est le plus petit ensemble de mots construits sur l'alphabet de la logique des prédicats tel que :

- Toute constante est un terme ;
- Toute variable est un terme ;
- $f(t_1,...,t_n)$ est un terme(dit fonctionnel) si f est une fonction à n arguments (n>0) et $t_1,...,t_n$ sont des termes.

Si P est un prédicat à n arguments (n>0) et $t_1,...,t_n$ sont des termes alors $P(t_1,...,t_n)$ (resp P) est une **formule atomique**.

B. Formules bien formées (fbf ou wff)

L'ensemble des formules de LP1 est le plus petit ensemble de mots construits sur l'alphabet de LP1 tel que :

- Si A est une formule atomique alors A est une formule;
- $\neg$ A est une formule si A est une formule;
- $(A \vee B)$, $(A \wedge B)$, $(A \Rightarrow B)$ et $(A \Leftrightarrow B)$ sont des formules si A et B le sont.
- (Q x A) est une formule si Q est un quantificateur, x une variable et A une formule.

Exemple: $\forall x\ A(x) \Rightarrow (\forall y\ P(f(x);a) \Rightarrow P(g(y); b))$) est une fbf

2.1.3.1. Occurrence et Portée

Définition 1 : Une occurrence d'une variable x dans une formule A est un endroit où x apparaît dans A sans être immédiatement précédée par $\forall$ ou $\exists$.

Sur une représentation en arbre, une occurrence liée de la variable *x* est une occurrence en dessous d'un sommet $\exists x$ ou $\forall x$. Une occurrence de *x* qui n'est pas sous un tel sommet est libre.

Définition 2 : Dans une formule A = Q x B, avec Q quantificateur et x variable , B est appelée la portée du quantificateur Q (Champs du quantificateur).

2.1.3.2 Variable Libre et variable Liée

a. Définition:

On dit qu'une variable est libre dans une formule si elle possède au moins une occurrence libre dans cette formule.

On dit qu'une variable est liée dans une formule si toutes les occurrences de la variable dans la formule sont liées.

Une formule sans variable libre(toute les variables sont liées) est appelée une formule fermée ou close.

b. Exemples :

1. F1= $(\forall X\ P(X)) \vee Q(X)$**.**

Dans cet exemple la variable 'X' est à la fois libre et liée .

2. F2= $\forall\ X\ A(X) \rightarrow [\ \forall\ Y\ (P(f(X),0) \rightarrow P(g(Y),b)\ ]$

Dans cet exemple:

- La première occurrence de X est liée
- La deuxième occurrence de X est libre

3. F3=$\forall X\ \exists Y\ (\ P(X,Y) \wedge \forall Z\ R(X,Y,Z))$

Dans cet exemple, F3 est une f**ormule fermée ou close**

2.1.3.3 Substitution et Unification

a. Substitution uniforme

Une substitution uniforme associe à une variable propositionnelle p une formule A. Elle est notée [p\A].

L'application de [p\A] à une formule B, notée $B_{[p\backslash A],}$ est le résultat du remplacement simultané de toutes les occurrences de p dans B par A. $B_{[p\backslash A],}$ est appelé une instance de B.

Exemple : $\sigma=[(x|f(a))(y|f(x))]$ est une substitution,

$F=P(x,f(x))$ est une formule,

D'où, $\sigma F= P(f(a),f(f(a)))$

b. Unification

Soit $F=\{A1,A2,\ldots An\}$ un ensemble fini de formules atomiques du calcul des prédicats du 1er ordre. On appelle unificateur de F toute substitution σ telle que:

$$\sigma A_1= \sigma A_2= \sigma A_3\ldots\ldots =\sigma A_n$$

Si σ est un unificateur d'un ensemble fini de formules F, alors pour toute substitution α: $\alpha\sigma$ est un unificateur de F.

Exemple: Soient les formules A1, A2 et A3 suivantes et σ une substitution telles que: $A_1=P(x,z)$

$A_2=P(f(y), g(a))$

$A_3= P(f(u),z)$

$\sigma=[(x|f(u)\ (y|u)\ (z|g(a))]$

Est ce que σ est un Unificateur ?

Solution:

$\sigma=[(x|f(u)\ (y|u)\ (z|g(a))]$

$A_1=P(x,z) \Rightarrow \sigma A_1= P(f(u),g(a))$

$A_2=P(f(y), g(a)) \Rightarrow \sigma A_2=P(f(u)),g(a))$

$A_3= P(f(u),z) \Rightarrow \sigma A_3=P(f(u),g(a))$

$\sigma A_1= \sigma A_2= \sigma A_3$

La substitution σ est donc un unificateur

2.1.4 Forme Prenexe et Forme Skolem

2.1.4.1 Définition 1 : Une formule A est en forme normale de Prénexe si elle est sans quantificateurs ou de la forme $Q_1x_1 \ldots Q_nx_n$ B , où B est une formule sans quantificateurs et les Q_i des quantificateurs ($\forall$ ou $\exists$).

La suite des quantificateurs est appelée préfixe, et B est appelée matrice.

2.1.4.2 Mise en forme Prenexe

Toute formule A est équivalente à une Forme Normale de Prenexe. Pour mettre une formule sous forme Prenexe, on applique l'algorithme suivant :

Algorithme de mise en forme normale de Prénexe (sans $\Leftrightarrow$ et $\Rightarrow$)

Entrée : une formule A1 **Sortie** : une FN de Prenexe A'1 équivalente à A1

Début **Eliminer** $\Leftrightarrow$ et $\Rightarrow$;

Appliquer à A1 les remplacements suivants autant de fois que nécessaire (Transférer les négations):

$\neg \neg A \rightarrow A$

$\neg (A \vee B) \rightarrow \neg A \wedge \neg B$

$\neg(A \wedge B) \rightarrow \neg A \vee \neg B$

Passage de ¬ devant les prédicats

$(\forall x\, A(x)) \wedge (\forall x\, B(x) \rightarrow \forall x\, (A(x) \wedge B(x))$

$(\exists x\, A(x)) \vee (\exists x\, B(x)) \rightarrow \exists\, x\, (A(x) \vee B(x))$

Sortie des quantificateurs devant les formules

Rendre A1 propre par renommage de variables.

Appliquer à A1 les remplacements suivants autant de fois que nécessaire

$(Q\, x\, A(x)) \wedge B \rightarrow Q\, x\, (A(x) \wedge B)$ (x est non libre dans B)

$(Q\, x\, A(x)) \vee B \rightarrow Q\, x\, (A(x) \vee B)$ (x est non libre dans B)

Fin

Figure 2.5. Algorithme de mise en forme normale de Prenexe

Exemple 1: Mettre sous forme de Prenexe

a. $F = \forall x\ (P(x) \rightarrow \exists x\ P(x))$

1. Eliminer $\rightarrow$: $\forall x(\neg P(x) \vee \exists x\ P(x))$
2. Renommer : $\forall x(\neg P(x) \vee \exists y\ P(y))$
3. Sortir $\exists y$: $\forall x\ \exists y\ (\neg P(x) \vee P(y))$ $\rightarrow$ Forme Normale de Prenexe

b. $G = (\forall x\ P(x)) \rightarrow (\exists x\ Q(x))$

1. Eliminer $\rightarrow$: $(\neg\ \forall x P(x)) \vee \exists x\ Q(x))$
2. Passage de $\neg$ devant les prédicats : $(\exists x\ \neg\ P(x) \vee \exists x\ Q(x))$
3. Sortir $\exists x$: $\exists x\ (\neg\ P(x) \vee Q(x))$ $\rightarrow$ Forme Normale de Prenexe

2.1.4.3 Définition 2:

Une formule est en forme normale de **Skolem** si elle est en forme normale Prenexe et ne contient pas de quantificateur existentiel.

La Skolémisation consiste a remplacer tous les quantificateurs existentiels dans une formule en forme normale de Prénexe, par:

- Des constantes sans quantification universelle avant.
- Sinon des fonction des variables quantifiées universellement.

On peut donc ignorer les quantificateurs, car toute variable est quantifiée universellement.

Propriétés de la Skolémisation : La skolémisation préserve la Satisfiabilité.

Exemple 2: Mettre la formule suivante sous forme de Skolem

$\exists x\ \exists y\ \forall z\ \forall t\ \exists v\ P(x,y,z,t,v)$

Solution:

Etape 1. x/a : $\exists y \; \forall z \; \forall t \; \exists v \; P(a,y,z,t,v)$

Etape 2. y/b : $\forall z \; \forall t \; \exists v \; P(a,b,z,t,v)$

Etape 3. v/f(z.t) : $\forall z \; \forall t \; P(a,b,z,t,f(z,t))$

$P(a,b,z,t,f(z,t))$ …….Forme Normale Skolem

Exemple 3: Soit la formule $A \equiv \exists x,(P(x) \Rightarrow \forall y, P(y))$.

1. Donner la forme de Skolem de A.
2. La formule A est-elle valide ? justifier la réponse

Solution:

1. $A \equiv \exists x,(\neg P(x) \vee \forall y, P(y))$

on introduit une constante a pour le connecteur existentiel et on obtient la formule

$\forall y, \neg P(a) \vee P(y)$

2. on procéde par équivalence

$A \equiv \exists x,(\neg P(x) \vee \forall y, P(y)) \equiv (\exists x,(\neg P(x)) \vee \forall y,$

$P(y) \equiv \neg(\forall x, P(x)) \vee \forall y, P(y)$ qui est de la forme $\neg C \vee C$ donc toujours vrai.

2.1.5 Introduction à la théorie de la démonstration

La théorie de la démonstration, (ou théorie de la preuve) est une branche de la logique mathématique, fondée par David Hilbert au début du $20^{ième}$ siècle.

En mathématiques, une démonstration est un raisonnement qui permet, à partir de certains axiomes, d'établir qu'une assertion est vraie. Un résultat qui est démontré s'appelle un théorème. Une fois le théorème démontré, il peut être utilisé comme base pour démontrer d'autres assertions.

Une démonstration consiste donc à formaliser un raisonnement par un système de preuves et en se basant sur un **système formel** qui définit des règles de calcul entre formules qui simulent le raisonnement, et ce qui permet de déduire de nouvelle formules.

Ce système est constitué par :

- Le langage et les formules sur ce langage ;
- Les axiomes : formules ou schémas de formules supposées vraies ;
- Les règles d'inférence : permettant de déduire de nouvelles formules.

Plusieurs Systèmes de Preuve sont utilisés tels la preuve à la Hilbert-Frege ; la preuve par Déduction naturelle et la preuve par Résolution et qui sont présentés ci-dessous.

2.2 Démonstration à la Frege et Hilbert

Les **systèmes à la Frege et Hilbert** servent à définir des *déductions* en suivant un modèle proposé par David Hilbert au début du 20^{ième} siècle : un grand nombre d'*axiomes logiques* exprimant les principales propriétés de la logique que l'on combine avec la règle "*Modus Ponens"*, pour dériver de nouveaux théorèmes. Ces systèmes constituent les premiers systèmes déductifs, avant l'apparition de la déduction naturelle.

La preuve à la Frege et Hilbert suit donc le principe suivant :

On part d'un ensemble d'axiomes, qui sont des tautologies et on utilise la règle de déduction du Modus Ponens. Le fondement de la règle du Modus Ponens est qu'à partir d' une formule 'F' et d'une formule 'F $\Rightarrow$ G', on peut déduire G.

La règle du Modus Ponens peut s'écrire :

$$\frac{F \qquad F \Rightarrow G}{G}$$ (Règle de Modus Ponens)

2.2.1 Ensemble d'Axiomes de la logique booléenne

Un axiome de la logique booléenne est n'importe quelle instance d'une des formules suivantes :

1. $(X_1 \Rightarrow (X_2 \Rightarrow X_1))$ (**axiome 1 pour l'implication**) ;
2. $((X_1 \Rightarrow (X_2 \Rightarrow X_3)) \Rightarrow ((X_1 \Rightarrow X_2) \Rightarrow (X_1 \Rightarrow X_3)))$ (**axiome 2 pour l'implication**) ;
3. $(X_1 \Rightarrow \neg\neg X_1)$ (**axiome 1 pour la négation**) ;
4. $(\neg\neg X_1 \Rightarrow X_1)$ (**axiome 2 pour la négation**) ;
5. $((X_1 \Rightarrow X_2) \Rightarrow (\neg X_2 \Rightarrow \neg X_1))$ (**axiome 3 pour la négation**) ;
6. $(X_1 \Rightarrow (X_2 \Rightarrow (X_1 \wedge X_2)))$ (**axiome 1 pour la conjonction**) ;
7. $((X_1 \wedge X_2) \Rightarrow X_1)$ (**axiome 2 pour la conjonction**) ;
8. $((X_1 \wedge X_2) \Rightarrow X_2)$ (**axiome 3 pour la conjonction**) ;
9. $(X_1 \Rightarrow (X_1 \vee X_2))$ (**axiome 1 pour la disjonction**) ;
10. $(X_2 \Rightarrow (X_1 \vee X_2))$ (**axiome 2 pour la disjonction**) ;
11. $(\neg X_1 \Rightarrow ((X_1 \vee X_2) \Rightarrow X_2))$ (**axiome 3 pour la disjonction**).

2.2.2 Principe de démonstration à la Hilbert-Frege

Soit T un ensemble de formules propositionnelles, et F une formule propositionnelle. Une preuve **à la Frege et Hilbert** (on dit aussi à la **Hilbert-Frege**) de F à partir de T est une suite finie $F_1, F_2, \ldots, F_n$ de formules propositionnelles telle que :

- F_n est égale à F,
- et pour tout i :
  - Ou bien F_i est dans T ;
  - Ou bien F_i est un axiome de la logique booléenne ;
  - Ou bien F_i s'obtient par Modus Ponens à partir de deux formules F_j et F_k avec $j < i$ et $k < i$.

On dit alors que F est prouvable à partir de T note $T \vdash F$

2.2.3 Exemple d'application

Démonstration en utilisant la preuve de Hilbert-Frege de (F ⇒ H) à partir de {(F ⇒ G), (G ⇒ H)}.

F1 : (G ⇒ H) (hypothèse) ;

F2 : ((G ⇒ H) ⇒ (F ⇒ (G ⇒ H))) (instance de l'axiome 1.) ;

F3 : (F ⇒ (G ⇒ H)) (Modus Ponens à partir de F1 et F2) ;

F4 : ((F ⇒ (G ⇒ H)) ⇒ ((F ⇒ G) ⇒ (F ⇒ H))) (instance de l'axiome 2) ;

F5 : ((F ⇒ G) ⇒ (F ⇒ H)) (Modus Ponens à partir de F3 et F4) ;

F6 : (F ⇒ G) (hypothèse) ;

F7 : (F ⇒ H) (Modus Ponens à partir de F6 et F5).

2.3 Démonstration par Déduction naturelle

La notion de démonstration Hilbert-Frege est en pratique difficile à utiliser En effet, dans le système précédent, on se contraint de garder les hypothèses tout au long de la démonstration. Elle s'automatise donc très mal.

La **déduction naturelle** est une méthode très proche du **raisonnement humain. Son principe est d'établir un chemin des hypothèses vers la conclusion**.

2.3.1 Règles de la Déduction naturelle

Pour établir la validité des formules, on introduit un système de déduction qui permet de déduire qu'une formule est une conséquence logique d'un ensemble de formules.

On écrit cette relation **Γ ⊢ P** avec Γ un ensemble de formules et P une formule. On appelle cet objet un **séquent.** L'interprétation de cette relation est que si on a pu dériver Γ ⊢ P et que les formules dans Γ sont vraies alors la formule P est vraie.

Ce système organise les étapes de déduction en deux catégories en fonction du connecteur principal d'une formule P :

1. Les étapes d'**Introduction (I)** expliquent comment on peut construire une dérivation de $\Gamma \vdash P$;
2. Les règles d' **Elimination** (**E**) expliquent comment on peut exploiter une preuve de $\Gamma \vdash P$ obtenue par ailleurs.

2.3.1.1 Principe de démonstration à la Déduction Naturelle (DN)

Une démonstration (Preuve) dans ce système (DN) se construit sous la forme d'un arbre dont :

- La racine est le Séquent (Formule) à prouver
- Chaque nœud correspond à une des règles de la logique.
- Les feuilles seront formées des règles qui n'ont pas de conditions (la règle d'hypothèse ou de la formule T).

Une formule est prouvée en déduction naturelle, si elle apparait en conclusion d'un arbre de preuve ne contenant plus aucune hypothèse temporaire.

2.3.1.2 *Règles de conjonction*

La règle d'introduction de la conjonction, notée $\wedge I$, signifie simplement que si nous avons une preuve de A et une preuve de B, alors nous pouvons déduire une preuve de la proposition $A \wedge B$.

Nous pouvons aussi dire que la preuve de la proposition $A \wedge B$ peut se décomposer en une preuve de A et une preuve de B.

$$\frac{A \quad B}{A \wedge B} \, I\wedge \qquad \text{(Règle 2.1)}$$

Les règles d'élimination de la conjonction, notées $\wedge E_1$ et $\wedge E_2$, permettent de déduire à partir de la conjonction de deux formules $A \wedge B$ soit la formule A soit la formule B. Ainsi nous avons éliminé le connecteur de la conjonction.

$$\frac{A \wedge B}{A} E_1\wedge \qquad \text{(Règle 2.2)}$$

$$\frac{A \wedge B}{B} E_2\wedge \qquad \text{(Règle 2.3)}$$

2.3.1.3 *Règles de disjonction*

Les règles d'introduction de la disjonction sont les duales des règles de l'élimination de la conjonction. Les deux règles d'introduction ($I_1\vee$ et $I_2\vee$) permettent de créer une disjonction de deux formules à partir d'une des deux formules. Ces règles indiquent que si A est vraie alors $A \vee B$ et $B \vee A$ sont aussi vraies quelle que soit la proposition B.

$$\frac{A}{A \vee B} I_1\vee \qquad \text{(Règle 2.4)}$$

$$\frac{A}{B \vee A} I_2\vee \qquad \text{(Règle 2.5)}$$

La règle d'élimination de la disjonction, notée $E\vee$, est plus complexe. Afin de déduire la formule C à partir de la disjonction $A \vee B$ il faut prouver les prémisses suivantes : A implique C et B implique C. Cette règle formalise la notion de preuve par cas: on souhaite prouver C alors deux cas , A ou B, sont possibles ; on se place dans le cas où A est vérifiée et on prouve C, puis on se place dans le cas où B est vérifiée et on prouve C.

$$\frac{A \vee B \qquad A \rightarrow C \qquad B \rightarrow C}{C} E\vee \qquad \text{(Règle 2.6)}$$

2.3.1.3 . *Règles de l'implication*

La règle de l'élimination de l'implication $E_{\rightarrow}$ dit que si on a une preuve de A et une preuve de $A \rightarrow B$ alors nous avons une preuve de B. Cette règle correspond à la règle du Modus Ponens.

$$\frac{A \qquad A \rightarrow B}{B} \; E\rightarrow \qquad \text{(Règle 2.7)}$$

Si nous pouvons déduire une formule B d'une hypothèse A, alors nous pouvons déduire $A \rightarrow B$ en nous passant de cette hypothèse: C'est la règle d'introduction de l'implication $I_{\rightarrow}$. Cette règle est résumée par le schéma ci-dessous, où la notation (i) indique que si A est une hypothèse de la preuve de B, cette hypothèse est enlevée de la preuve de $A \rightarrow B$.

$$\frac{\begin{array}{c} A^{(i)} \\ \vdots \\ B \end{array}}{(A \rightarrow B)} \; I\rightarrow \; \cancel{(i)} \qquad \text{(Règle 2.8)}$$

2.3.1.4 . *Règle du "Faux"*

Notée E f q, pour la formule latine « ex falso, quodlibet », cette règle indique que du faux, on peut déduire ce qu'on veut.

$$\frac{\perp}{A} \; Efq \qquad \text{(Règle 2.9)}$$

2.3.1.5 . *Règle de "Réduction à l'absurde***"**

Notée RAA pour « Reductio Ad Absurdum », Cette règle élimine deux occurrences successives de la négation.

$$\frac{\neg \neg A}{A} \; RAA \qquad \text{(Règle 2.10)}$$

La figure ci-dessous présente les principales règles de déduction:

Elimination		**Introduction**	
$\frac{(A \wedge B)}{A} \quad \frac{(A \wedge B)}{B}$	$(E\wedge)$	$\frac{A \quad B}{(A \wedge B)} \quad \frac{A \quad B}{(B \wedge A)}$	$(I\wedge)$
$\frac{(A \vee B) \quad \begin{matrix} A^{(i)} & B^{(j)} \\ \vdots & \vdots \\ C & C \end{matrix}}{C}$	$(E\vee)$	$\frac{A}{(A \vee B)} \quad \frac{B}{(A \vee B)}$	$(I\vee)$
$\frac{(A \rightarrow B) \quad A}{B}$	$(E\rightarrow)$	$\frac{\begin{matrix} A^{(i)} \\ \vdots \\ B \end{matrix}}{(A \rightarrow B)}\,(i)$	$(I\rightarrow)$
$\frac{(A \leftrightarrow B) \quad A}{B} \quad \frac{(A \leftrightarrow B) \quad B}{A}$	$(E\leftrightarrow)$	$\frac{\begin{matrix} A^{(i)} & B^{(j)} \\ \vdots & \vdots \\ B & A \end{matrix}}{(A \leftrightarrow B)}\,(ij)$	$(I\leftrightarrow)$
$\frac{\bot}{A}$	$(E\bot)$	$\frac{A \quad \neg A}{\bot}$	$(I\,\bot)$
$\frac{\begin{matrix} \neg A^{(i)} \\ \vdots \\ \bot \end{matrix}}{A}\,(i)$	$(E\neg)$	$\frac{\begin{matrix} A^{(i)} \\ \vdots \\ \bot \end{matrix}}{\neg A}\,(i)$	$(I\,\neg)$

Figure 2.6. Règles de déduction

2.3.2 Validité et complétude

Théorème de Validité. *Pour tout ensemble de formules Γ et pour toute formule A, si Γ ⊢ A est prouvable, alors A est une conséquence de Γ.*

Théorème de Complétude. *Soit Γ un ensemble de formules. Soit A une formule qui est une conséquence de Γ, alors Γ ⊢ A est prouvable.*

Donc , on dit que F est prouvable à partir de T (T ⊢ F) si et seulement si T ⊨ F *avec*

- ***T ⊨ F*** *: signifie que tout modèle de T est modèle de F ou F est une conséquence de T.*
- ***T ⊢ F****: F est prouvable à partir de T (déduite).*

2.3.3 Exemples d'application

1. Démontrer par Déduction Naturelle (DN) que : (A∧B)→ A

Solution:

$$\frac{\dfrac{(\mathrm{A}\wedge\mathrm{B})^{(1)}}{\mathrm{A}}\,(\mathrm{E}\wedge)}{(\mathrm{A}\wedge\mathrm{B})\rightarrow\mathrm{A}}\,\mathrm{I}\rightarrow\ (1)$$

2. Démontrer par Déduction Naturelle (DN) que : **(a∧b)→ (b∧a)**

Solution :

$$\frac{\dfrac{\dfrac{a\wedge b^{(1)}}{b}E\Lambda\quad\dfrac{a\wedge b^{(1)}}{a}E\Lambda}{(\mathrm{b}\wedge a)}}{(a\wedge b)\rightarrow(b\wedge a)}\ I\rightarrow(1)$$

2.4 Démonstrations par Résolution

C'est une règle inventée par Robinson (1965) Elle est fortement utilisée dans les systèmes de preuve automatique (elle est à la base du langage de programmation logique Prolog.). Dans cette démonstration, une seule règle qui utilise uniquement des clauses est à appliquer. Il faut donc mettre la formule sous forme FNC (Forme Normale Conjonctive).

2.4.1 Règle de Résolution

Théorème. *Le schéma suivant est bien une règle d'inférence :*
$\{X \vee A, \neg X \vee B\} \models A \vee B$ (règle de résolution)

Démonstration: $(\neg X \vee B) \equiv (X \Rightarrow B)$ et que $(A \vee X) \equiv (\neg A \Rightarrow X)$, on a ainsi: $\{\neg A \Rightarrow X, X \Rightarrow B\}$, qui donne par règle du syllogisme: $\neg A \Rightarrow B$, équivalent à $A \vee B$.

2.4.2 Résolvante de deux clauses

Soit deux clauses S1 et S2 telles que le littéral L figure dans l'une et le littéral $\neg$L dans l'autre (on écrit $L \in S1$, $\neg L \in S2$).

On appelle Résolvante de S1 et S2 notée Res(S1,S2), la clause obtenue en supprimant le littéral L de l'une et le littéral $\neg$L de l'autre et réunissant tout ce qui reste en une seule clause (On supprimera les littéraux redondants).

Exemple : $S1 = \neg p \vee q \vee r$ et
$S2 = p \vee q \vee s$
$Res(S1,S2) = q \vee r \vee s$

2.4.1 Principe de Résolution

Une preuve par résolution de la clause F à partir de l'ensemble de clauses C est une suite finie de clauses C1,C2, …Cn telles que Ci ∈ C ou bien Ci est une résolvante de deux Cj précédentes, et Cn = F. Notation : C⊢ F

2.4.2 Résolution par réfutation

Une réfutation d'un ensemble de clauses C est une preuve de la clause vide à partir de N. Notation : C ⊢ □

2.4.3 Validité et complétude de la méthode de résolution

Corollaire 1: Toute clause F déduite d'un ensemble C de clauses par résolution est conséquence sémantique de C.

Si C ⊢ F alors C⊨ F

La règle de résolution est dite alors ***saine*** .

Corollaire 2: Un ensemble de clauses admettant une réfutation est contradictoire.

Si C ⊢ □ alors C⊨ f

Théorème. *La méthode de résolution est complète pour la réfutation.*

Si C est un ensemble de clauses contradictoires alors il existe une réfutation de C.

2.4.4 Exemples d'application

Exemple 1 : {p∨q∨r, ¬p∨q∨r, ¬q∨r} ⊢ r

Ensemble de départ : {p∨q∨r, ¬p∨q∨r, ¬q∨r}

1er pas: {**p∨q∨r**, **¬p∨q∨r**, ¬q∨r, q∨r}

2eme pas : {p∨q∨r, ¬p∨q∨r, **¬q∨r**, **q∨r**}

3eme pas : {p∨q∨r, ¬p∨q∨r, **¬q∨r**, **q∨r**, r}

stop! on a trouvé r.

Exemple2 : {p∨q∨r, ¬p∨q∨r, ¬q∨r} ⊢ r

Ensemble de départ: {p∨q∨r, ¬p∨q∨r, ¬q∨r, ¬r}

1er pas: {p∨q∨r, ¬p∨q∨r, **¬q∨r**, **¬r**, ¬q}

2eme pas : {**p∨q∨r**, **¬p∨q∨r**, ¬q∨r, ¬r, ¬q, **q∨r**}

***3eme pas*:** {p∨q∨r, ¬p∨q∨r, ¬q∨r, ¬r, **¬q**, **q∨r**, r}

4eme pas: {p∨q∨r, ¬p∨q∨r, ¬q∨r, **¬r**, ¬q, q∨ r, **r**, □}

stop! on a trouvé □ (□ :clause vide).

Exemple 3:

De l'ensemble de clauses (1), (2), (3) et (4):

(1) ¬p∨¬q∨r

(2) ¬p∨q

(3) p

(4) ¬r

On obtient successivement :

(5) ¬p∨¬q a partir de (1) et (4)

(6) ¬p a partir de (2) et (5)

(5) □ a partir de (3) et (6)

Conclusion

La logique est une base essentielle pour justifier le raisonnement. C'est pour cette raison que l'introduction a été dédiée aux concepts de la logique propositionnelle et des prédicats.

Une démonstration est un ensemble structuré d'étapes correctes du raisonnement. Dans une démonstration, chaque étape est soit un *axiome (proposition vraie)*, soit l'application d'une règle qui permet d'affirmer qu'une proposition, la *conclusion*, est une conséquence logique d'une ou plusieurs autres propositions. Les prémisses sont soit des axiomes, soit des propositions déjà obtenues comme conclusions de l'application d'autres règles. La conclusion de l'étape ultime d'une démonstration est un *théorème*.

Ce chapitre a introduit les éléments de base de la logique : calcul propositionnel et calcul des prédicats du premier ordre. Les concepts et notions présentées dans cette partie introductive sont des pré-requis essentiels pour l'utilisation des méthodes de démonstrations. Additionnement, les concepts liés à la théorie de preuve ou démonstration ont été exposés. Les principaux systèmes formels de démonstration (Preuve à la Frege et Hilbert, Déduction Naturelle et Preuve par résolution) ont été aussi détaillés. Dans le chapitre suivant, on détaillera les modèles de calculs.

Chapitre 3

Modèles de Calculs

Chapitre 3

Modèles de Calculs

Contenu du chapitre 3

Introduction

David Hilbert, Parrain des mathématiques a posé le problème de la décision : Existe –t-il un algorithme permettant de déterminer si un énonce et démontrable dans un système formel données ? (Preuve= Algorithme). Plusieurs solutions formelles pour la notion de calcul ont été proposées suite à ce problème de Hilbert, notamment, les fonctions récursives (Herbrand, Gödel, 1930), le λ-calcul (Church, 1930) et notamment ce qui va être présenter dans cette partie du cours, les machines de Turing (proposées par Alain Turing en 1936).

Ce chapitre a été rédigé à partir des ouvrages [Stern,1994], [Wolper, 2001], [Hopcroft, 2001], [Carton, 2008] et [Bournez, 2021].

3.1 Machines de Turing

La machine de Turing est un modèle de machine abstraite introduit en 1936 par le chercheur anglais Alan Turing dans un article fondateur intitulé "*On Computable Numbers, With An Application To The Entscheidungsproblem (problème de décision)*", dans lequel il propose une réponse à une **question de décidabilité** posée huit ans plus tôt par le célèbre mathématicien David Hilbert. En effet ce terme de décidabilité se voit proposer plusieurs définitions exposées ci-dessous :

Définition 1 : Un énonce est **décidable** dans un système formel donné s'il peut être **démontré (Preuve)** dans ce système. Il est **indécidable** sinon.

Définition 2 : Le système formel est **décidable** s'il **existe un algorithme** permettant de savoir si un énoncé donné est décidable ou pas.

Définition 3 : Un problème est (algorithmiquement) **décidable** s'il existe une **machine de Turing qui calcule la fonction associée**. Il est indécidable sinon.

La dernière définition présente la machine de Turing comme un outil de vérification. En fait, les machines de Turing sont des objets mathématiques pour décrire les algorithmes. Ils sont ainsi largement utilisés en théorie de la calculabilité, en théorie de la complexité et en théorie de l'approximation. La raison principale tient à son extrême simplicité qui a permis d'établir des résultats, beaucoup plus difficiles à démontrer avec des modèles moins rudimentaires.

En effet, la machine de Turing est le modèle le plus simple que l'on puisse concevoir et qui satisfait aux critères informels mais universels qui caractérisent un algorithme *:* déterminisme, discrétion, finitude, généralité.

Nous présentons ci-dessous les composants essentiels d'une machine de Turing.

3.1.1 Ingrédients

Les automates représentent un modèle de calcul utilisant une mémoire finie correspondant à l'ensemble des états de la machine. D'autre part, les machines de Turing sont également des machines "finies" (correspondant à des algorithmes effectifs), mais elles utilisent une mémoire plus élaborée. Elles utilisent un (ou des) ruban "infini" de cases mémoires qu'elles lisent et réécrivent.

Une machine de Turing est donc composée des éléments suivants :

- Une mémoire infinie sous forme de ruban, divisée en cases,
- Chaque case peut contenir un élément d'un alphabet,
- Une tête de lecture qui se déplace sur le ruban
- Une fonction de transition qui pour chaque état de la machine q, donne selon le symbole lu :
    - L'état q' suivant ;
    - Le symbole à écrire sur le ruban;
    - Un sens de déplacement pour la tête de lecture à gauche ou à droite.

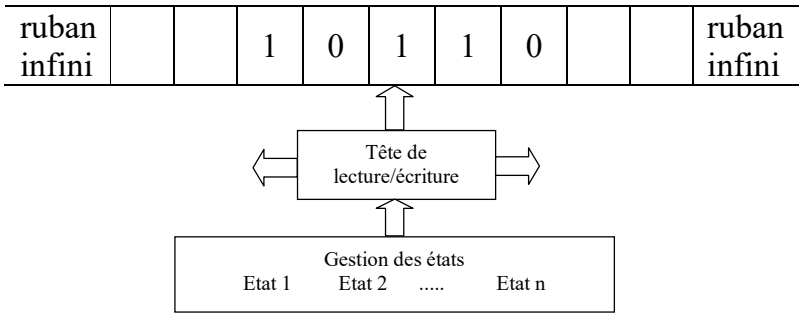


Figure 3.1 Machine de Turing : Ruban avec tête de lecture

A chaque étape de l'exécution, la machine, selon son état, lit le symbole se trouvant sous la tête de lecture, et selon ce symbole :

- Remplace le symbole sous la tête de lecture par celui précisé par sa fonction transition ;
- Déplace possiblement cette tête de lecture d'une case vers la droite ou vers la gauche suivant le sens précisé par la fonction de transition ;
- Change d'état vers l'état suivant.

3.1.2 Description

Il n'existe pas de description formelle unique pour une machine de Turing. On présente ci-dessous une définition :

Une machine de Turing déterministe est un t-uplet $(Q, \Sigma, \Gamma, \#, \delta, q_0, F)$ ou :

- Q est un ensemble fini d'états,
- Σ est l'alphabet d'entrée,
- Γ est l'alphabet de travail,
- # est le symbole blanc (noté quelque fois B, \$,□),
- δ est la fonction de transition telle que $\delta : Q \times (\Sigma \cup \Gamma \cup \{\#\}) \rightarrow Q \times (\Sigma \cup \Gamma \cup \{\#\}) \times \{\leftarrow, \rightarrow\}$,
- $q_0 \in Q$ est l'état initial,
- $F \subseteq Q$ est l'ensemble des états.

Il existe d'autres définitions formelles de Machines de Turing qui spécifient les états d'acceptation q_a et de refus q_r (ou d'arrêt).

Exemple 1: $(Q, \Sigma, \Gamma, \#, \delta, q_0, F) = (\{0,1,2\},\{a,b\},\{A\},\#, \delta, 0, \{2\})$

- $Q = \{0,1,2\}$,
- $\Sigma = \{a,b\}$,
- $\Gamma = \{A\}$,
- Symbol blanc = #,
- $F = \{2\}$,
- $\delta(0,a) = (1,A,\rightarrow)$,
- $\delta(1,b) = (2,\#,\leftarrow)$,
- $\delta(2,\#) = (2,\#,\leftarrow)$,
- $\delta(2,A) = (2,\#,\leftarrow)$.
-

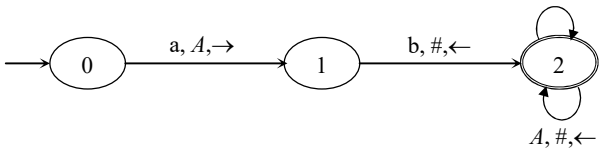


Figure 3.2 Exemple de Machine de Turing

a. Langage accepté/ refusé par Machine Turing (MT)

Définition : Un langage $L \subseteq \Sigma^*$ est *récursif* , s'il existe une machine de Turing M telle que :

1. L(M) = L
2. M s'arrête pour tout mot dans Σ^*.

On dit que L est *décidé* par M.

Remarque :

- Un langage est accepté si tous ses mots sont acceptés.
- On dit que la machine de Turing boucle sur un mot w, si w n'est ni accepté, et ni refusé.

b. Configuration

Une configuration est donnée par la description du Ruban, par la position de la tête (Lect./Ecrit.) et par l'état interne.

C'est un triplet comportant (u,q,v) ou' $q \in Q$ et $u,v \in (\Sigma \cup \Gamma)^*$:

- l'état q de la machine ,
- le mot u apparaissant avant la tête,
- le mot v se trouvant entre la tête et le dernier symbole non blanc.

Exemple 2 :

La configuration est (ab,q,bbab) :

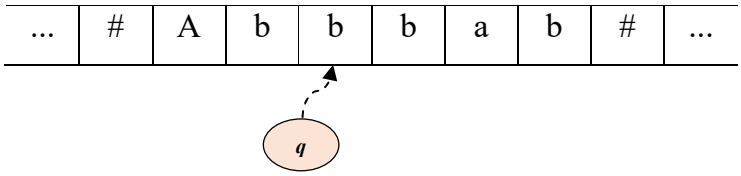


Figure 3.3 Exemple de configuration

- Une configuration initiale pour v est (ε,q_0,v) ou q_0 est l'état initial et est le mot vide. Une configuration finale est une configuration de la forme (u,q,v) avec $q \in F$.
- Deux Configurations C1 et C2 sont dites consécutives ($C1 \vDash C2$) si elles correspondent à l'activation d'une transition dans la machine de Turing.
- Un calcul sur un mot est une suite de configurations consécutives partant de la configuration initiale.

Exemple 3

(ab,q_1,bbab) (ab,q_1,bbab) |= (ab#,q_2,bab)

Figure 3.4 : Exemple de configurations consécutives

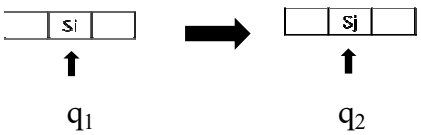
3.1.3 Programmer avec des machines de Turing

La machine de Turing est un modèle universel de calcul. Elle peut calculer tout ce que n'importe quel ordinateur physique peut calculer (aussi puissant soit-il). Inversement, ce qu'elle ne peut pas calculer ne peut l'être non plus par un ordinateur. Elle résume donc de manière saisissante le concept d'ordinateur et constitue un support idéal pour raisonner autour de la notion d'algorithme de calcul ou de démonstration.

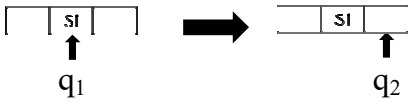
Toutefois, la programmation avec des machines de Turing est extrêmement bas niveau. En effet, programmer avec des MT consiste a définir le triplet {S,E,I}tel que:

- S : ensemble des symboles finis de la MT, S = {S_0, S_1, .. S_n}
- E : Ensemble des états internes de la MT, S = {q_0, q_1, .. q_f}
- I : Ensemble des instructions de la MT. On a trois types d'instruction :

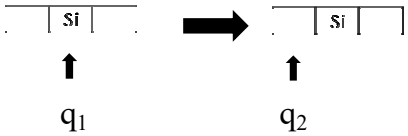
1. Ecriture de symbole S j : $q_1 S_i\ S_j q_2$



2. Déplacement de la tête d'une case à droite : $q_1 S_i D q_2$



3. Déplacement de la tête d'une case à gauche : $q_1 S_i G q_2$



3.1.4 Techniques de programmation

Alan Turing inventa cette machine abstraite en 1936 pour définir la notion de "fonction calculable". En effet, toute tâche exécutée par une machine de Turing MT peut l'être sur un ordinateur et réciproquement.

Soit F une fonction de N→N, MT une Machine de Turing déterministe. La fonction F est dite calculable par la machine de Turing si étant donnée la configuration initiale : $q_0 \underline{n_1} * \underline{n_2} * \underline{n_3} .. * \underline{n_p}$ et en appliquant les instruction I de MT, on arrive a une situation finale avec la configuration finale $q_f \, \underline{F(n_1, n_2, .. n_p)}$.

On pose donc : $q_0 \underline{n_1} * \underline{n_2} * \underline{n_3} .. * \underline{n_p} \rightarrow q_f \, \underline{F(n_1, n_2, .. n_p)}$

Remarque :

- Une même fonction peut être calcule par plusieurs MT différentes.
- Une même MT peut calculer plusieurs fonctions différentes.

La représentation d'un entier n est notée $\underline{n}$. Ce dernier est représenté par n+1 barre (pour pouvoir représenter le nombre 0).

$$\underline{0} =_{def} 1 = 1^1$$
$$\underline{1} =_{def} 11 = 1^2$$
$$\underline{2} =_{def} 111 = 1^3$$
$$\vdots$$

$\underline{n} =_{def}$ | 0 | 1 | 1 | ... | 1 | 0 | $\underline{n} =_{def} 11....1 = 1^{n+1}$

q_0

Pour représenter les n-uplets, on utilise le symbole * pour séparer les nombres.

$$(n,m) =_{def} \underline{n}*\underline{m}$$
$$= 1^{n+1}*1^{m+1}$$

$$(n_1,n_2,n_3,.....,n_p) =_{def} \underline{n_1}*\underline{n_2}*n_3*....*n_p$$
$$= 1^{n_1+1}*1^{n_2+1}*1^{n_3+1}... *1^{n_p+1}$$

$(\underline{n},\underline{m}) =$ | 0 | 1 (n+1) fois: 1 | 1 | ... | 1 | * | 1 (m+1) fois: 1 | 1 | ... | 1 | 0 | $_{def}$

q_0

Il existe deux techniques utilisées couramment dans la programmation des machines de Turing :

- La première consiste à coder une information finie dans l'état de la machine. Pour cela l'information à stocker doit être finie.
- Une seconde technique consiste en l'utilisation de sous-procédures qui correspondent en fait à des collages de morceaux de programme au sein du programme d'une machine.

3.1.5 Application

1. Programmer avec une machine de Turing (MT) la fonction "successeur" $f:x \rightarrow x+1$

$$q_0\, \underline{n} \rightarrow q_f\, \underline{n+1}$$
$$\text{ou} \quad q_0\, 1^{n+1} \rightarrow q_f\, 1^{n+2}$$

Solution

$q_0\ 1\ G\ q_0$

$q_0\ 0\ 1\ q_f$

2. 1. Programmer avec une machines de Turing (MT) la fonction "nulle" f:x→0

$$q_0\ \underline{n} \quad \rightarrow \quad q_f\ \underline{0}$$

$$\text{ou} \quad q_0\ 1^{n+1} \rightarrow q_f\ 1$$

Solution

$q_0 1 0 q_1$

$q_1 0 D q_0$

$q_0 0 1 q_f$

3.1.6 Variantes de la notion de machine de Turing

a. Restriction à un alphabet binaire

Proposition 1 : Toute machine de Turing qui travaille sur un alphabet Σ quelconque peut être simulée par une machine de Turing qui travaille sur un alphabet Σ = Γ avec uniquement deux lettres (sans compter le caractère blanc).

On peut alors transformer le programme d'une machine de Turing M qui travaille sur l'alphabet Σ en un programme M1 qui travaille sur un codage binaire.

b. Machines de Turing à plusieurs rubans

On peut aussi considérer des machines de Turing avec plusieurs rubans (k rubans). Chacun des k rubans possède sa propre tête de lecture avec un nombre fini d'états Q. La fonction de transition δ est une fonction de $Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{\leftarrow, \rightarrow\}^k$.

En fonction de l'état de la machine et de ce qui est lu en face des têtes de lecture de chaque ruban, la fonction de transition donne les nouveaux symboles à écrire sur chacun des rubans, et les déplacements à effectuer sur chacun des rubans.

Proposition 2 : Toute machine de Turing qui travaille avec k-rubans peut être simulée par une machine de Turing avec un unique ruban.

3.1.7 Localité de la notion de calcul

Une spécification des diagrammes espace-temps d'une machine donnée : un tableau est un diagramme espace-temps de M sur une certaine configuration initiale C_0 si et seulement si d'une part sa première ligne correspond à C_0, et d'autre part dans ce tableau, le contenu de tous les rectangles possible sont parmi les fenêtres légales (fenêtres légales: contenus possibles pour la machine M).

3.2 Machines de Turing non-déterministes

Une autre généralisation naturelle des machines de Turing est d'avoir un nombre fini de transitions possibles à chaque pas de calcul. Il est assez facile de montrer que de telles machines peuvent être simulées par des machines déterministes. Pour cela on énumère au fur et à mesure l'ensemble des configurations possibles sur un ruban auxiliaire.

Une machine de Turing non-déterministe est caractérisé par le fait qu'à partir d'un état et un symbole courant, il peut y avoir plusieurs choix de comportements possibles. Une telle machine est décrite par $M = (K, \sum, \Gamma, \Delta, q_0, F)$ où : Δ est un sous-ensemble de $K \times \Gamma \times K \times \Gamma \times \{\leftarrow, \rightarrow, S\}$.

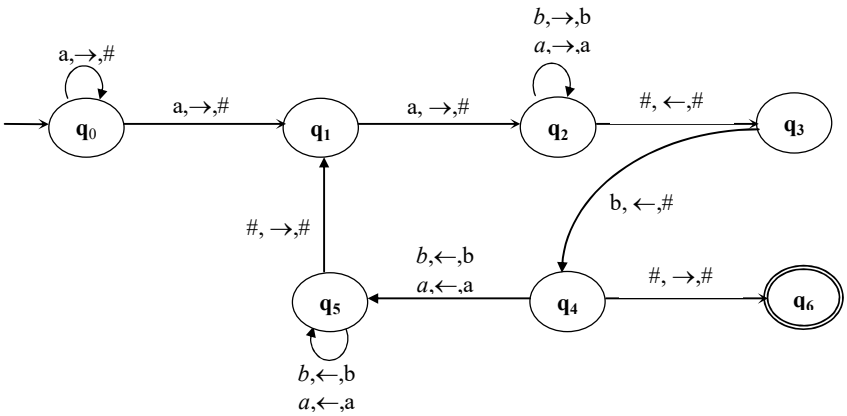


Figure 3.5 Exemple de Machine de Turing non-déterministe

Ainsi une machine de Turing non déterministe peut produire deux sorties différentes pour une même entrée. Une machine non déterministe est donc un accepteur dont le seul résultat qui nous intéresse est de savoir si la machine s'arrête ou non, sans considérer le contenu du ruban.

Le non déterminisme n'apporte aucune puissance supplémentaire. En effet, pour toute machine de Turing non déterministe M, on peut construire une machine normale M' telle que pour toute chaîne w $\in \sum^*$ on a:

- si M s'arrête avec w en entrée, alors M' s'arrête sur w,
- si M ne s'arrête pas sur l'entrée w, alors M' ne s'arrête pas non plus sur w.

Théorème. *Tout langage accepté par une machine de Turing non déterministe est accepté par une machine de Turing déterministe*

A l'aide de telle machine, tout automate à pile (non nécessairement déterministe) peut-être représenté par une machine de Turing. En particulier les langages algébriques sont bien décidables (au sens des machines de Turing).

3.3 Modèles rudimentaires

3.3.1 Machines à piles ($k \geq 2$)

Toute machine de Turing peut être simulée par une machine à 2 piles. En effet,une configuration d'une machine de Turing, vu précédemment, correspond à C =(u,q,v), où u et v désignent le contenu respectivement à gauche et à droite de la tête de lecture du ruban.

Le passage d'une configuration a une autre (Configurations consécutives) peut se traduire par des opérations 'empiler', 'dépiler'. On peut donc voir u et v comme des piles et la machine de Turing a une machine à deux piles.

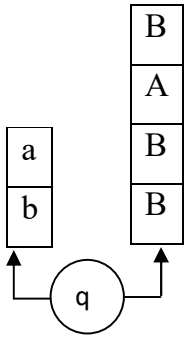


Figure 3.6 Machine de Turing de l'exemple 2 vue comme une machine à 2 piles.

3.3.2 Machines à compteurs

Les machines à compteurs sont généralement considérées comme des machines qui calculent des fonctions sur les entiers, ou comme reconnaissant des sous-ensembles de n-uplets d'entiers. Si l'on veut calculer sur les mots sur un alphabet ($\Sigma=\{0,1\}$. Ainsi, il est nécessaire de coder les mots comme des entiers.

Une machine à compteurs possède un nombre fini de compteurs. Ses instructions permettent d'incrémenter (Inc) ou de décrémenter (Decr) un compteur ou encore tester l'égalité à zéro (IsZero). Initialement, les compteurs ont initialisé à zéro sauf ceux qui codent une entrée quelconque.

3.4 Thèse de Church-Turing

Les fonctions Calculables par quelque moyen que ce soit (Mécanique ou Mental) sont celles que l'on peut représenter par une machine de Turing

Thèse de Church/Turing : Une fonction est calculable si et seulement si elle est calculable par machine de Turing.

Remarque : Tous les modèles connus (notions de machine, langages de programmation…) peuvent être représenter par des machines de Turing.

3.4.1 Équivalence de tous les modèles considérés

Après avoir introduit les machines de Turing, les chercheurs ont montré que les différentes variantes de ces machines conduisaient toutes à la même notion de calculabilité. De même, les fonctions récursives sont équivalentes aux machines de Turing.

Ainsi, une fonction est récursive si et seulement si elle est calculable par une machine de Turing. Beaucoup d'autres modèles de calcul comme le λ-calcul ou les machines RAM ont été introduits. Et chacun de ces modèles s'est avéré équivalent aux machines de Turing.

Ceci a conduit Church à postuler que tous les modèles de calcul définissent la même notion de calculabilité. Ce postulat ne peut bien sûr pas être démontré mais il est universellement admis.

Conclusion

Les modèles de calcul cherchent à répondre à la question "Qu'est-ce qui est calculable ?". Pour cela, les théoriciens ont cherché à modéliser le processus du calcul via des modèles formels qui sont à la base de plusieurs théories. Ces modèles permettent d'appréhender les limites conceptuelles du calcul informatique. Ils modélisent formellement la notion de calcul effectif ou encore celle d'algorithme. Dans ce chapitre, le principe de fonctionnement des machines de Turing a été présenté suivi par une présentation des modèles rudimentaires. Finalement la thèse de Church-Turing a été exposée. Dans le chapitre suivant, on abordera les notions de machines universelles et d'interpréteurs .

Chapitre 4

Calculabilité

Chapitre 4
Calculabilité

Contenu du chapitre 3

Introduction

La thèse de Church-Turing a été reformulée par Lewis – Papadimitriou comme suit: "Un algorithme est une machine de Turing qui s'arrête pour toutes ses entrées". Jusqu'à présent toutes les notions d'algorithmes produites par l'homme sont équivalentes à celle-ci. Personne ne pense que c'est possible de contredire cette thèse.

*D'autre part, l*a *calculabilité cherche à identifier la classe des fonctions qui peuvent être calculées à l'aide d'un algorithme. Une bonne appréhension de ce qui est calculable et de ce qui ne l'est pas permet de voir les limites des problèmes que peuvent résoudre les ordinateurs. Dans ce dernier chapitre, les concepts de machines universelles, d'interpréteurs, de calculabilité et de décidabilité vont être abordés.*

Ce chapitre a été rédigé à partir des ouvrages [Wolper, 2001], [Hopcroft, 2001], [Carton, 2008] , [Bournez, 2021]et [Prost, 2021].

4.1 Machines universelles

Soit M une machine de Turing quelconque et x une entrée, la sortie de M sur l'entrée x est notée M(x). Une machine U est universelle si l'on peut écrire une entrée y sur le ruban telle que le calcul de U sur y donne M(x) en sortie.

En effet M a une description finie qui est sa table de transition. Cette description peut être écrite sur le ruban sur n cellules. L'entrée x peut s'écrire sur d'autres cellules vides. Il est possible donc de construire une machine U qui lit l'entrée x sur le ruban, ensuite lit la table de transition de M sur les n cellules dédiées du ruban. Une telle machine U est appelée Machine Universelle.

Définition : Une machine de Turing universelle U est une machine capable de simuler tout autre machine de Turing avec n'importe quelle entrée.

$$\forall P \; \exists \; A \; / \; A \text{ résout } (P) \rightarrow U \text{ résout } (P)$$

P : Problème A : Algorithme U : Machine Universelle

Cette définition fait le fondement des ordinateurs d'aujourd'hui (Informatique théorique): et reflète les concepts de **'Décidabilité et Calculabilité'.**

Les machines de Turing sont une réponse à la recherche de définition d'une fonction calculable. En effet, toute fonction calculable est calculable grâce à une machine de Turing. C'est la thèse de Church : "Toute fonction physiquement calculable est calculable par une machine de Turing".

Les fonctions calculables par d'autres modèles de calculs suffisamment riches (machines à registres, machine RAM, lambda calcul...) sont les mêmes que les fonctions calculables par des machines de Turing.

D'autre part, une machine de Turing universelle simule toutes les autres machines de Turing. Les machines de Turing sont en fait des modèles universels de calcul.

Remarque: De nos jours, un ruban est appelé disque dur ou mémoire, la table de transition (action) de M écrite sur le ruban est un programme, et U est un ordinateur .. !

4.1.1 Interpréteurs

Une machine de Turing universelle U peut être vue comme un interpréteur de machines de Turing. En effet, comme les interpréteurs, elle prend en paramètre une autre machine de Turing que l'on applique un mot d'entrée ω. La machine universelle donne alors le résultat de la machine qu'elle simule sur son entrée ω. En clair construire une machine de Turing qui serait un interpréteur de machines de Turing

4.2 Principe de fonctionnement de la machine de Turing Universelle U.

Les raisonnements reposent sur l'utilisation d'une machine de Turing universelle U capable d'exécuter toute machine de Turing M sur un mot ω. Le codage permet de représenter par un mot binaire m, une machine de Turing M ie. $m = [M]_2$.

Ainsi, la machine Universelle U est une machine de Turing à deux bandes qui prend en premier argument la représentation binaire d'une machine de Turing M inscrite sur la bande B_2 sous la forme d'un mot binaire m, puis prend en second argument un mot $\omega \in \{0,1\}$ inscrit sur la bande B_1.

Cela donne le théorème suivant :

Théorème. *Il existe une machine de Turing U telle, que sur l'entrée (M, ω) où :*

1. *$[M]_2$ est le codage de la machine de Turing M ;*
2. *$\omega \in \{0,1\}$;*

U simule la machine de Turing M sur l'entrée ω.

4.2.1 Représentation binaire d'une machine de Turing

Soit M donc une machine de Turing que l'on applique sur un mot d'entrée ω et U une machine universelle: U("M" "w") = "M(w)"

" " : codage de la machine M dans le langage de la machine U

M = (K, Σ, Γ, δ, s, H)

On doit coder une machine dans U par ΣU

$\Sigma \rightarrow \#_M, D_M, G_M : \underbrace{a\ldots\ldots}_{\text{nombre binaire}}..$

$\#_M$: a00…000
D_M : a00…001
G_M : a00…010

K → q1 : q00…000
q2 : q00…001
…

Une machine de Turing est alors représentée par sa table de transitions : "M" : suite de chaînes ("q","a","p","b","s") rangées dans l'ordre lexicographique croissant

Exemple:

Soit la machine de Turing M = ($\{q_0,q_1,q_2\}$, {a,b}, {a,b,#}, δ, q_0) et telle que:

- $\delta(q_0,\#) = (q_1,\#,\rightarrow)$;
- $\delta(q_1,\#) = (q_2,\#,\rightarrow)$;
- $\delta(q_2,a) = (q_2,a,\leftarrow)$;
- $\delta(q_1,a) = (q_1,b,\rightarrow)$;
- $\delta(q_1,b) = (q_1,a,\rightarrow)$;
- $\delta(q_2,b) = (q_2,b,\leftarrow)$.

La codification est la suivante:

q_0 : q000
: a000
→ : a011
q_1 : q001
q_2 : q010
a : a001
b : a010
← : a100

Le codage se fera de la façon suivante:

"M" est représentée par les transitions δ :

(q000, a000,q001,a000,a011),

(q000, a000,q010,a010,a100),

(q001, a001,q001,a010,a011),

(q001, a010,q001,a001,a011)

(q010, a001,q010,a001,a100),

(q010, a010,q010,a010,a100)

4.2.2 Schéma fonctionnel d'une machine de Turing universelle

On peut construire une machine de Turing universelle U permettant de simuler le fonctionnement d'une machine de Turing quelconque T. L'idée permettant de construire une telle machine est la suivante:

1. La table de transition de la machine T à simuler est codée sous la forme d'une séquence binaire.

2. La bande de la machine U universelle est divisée en trois parties distincte: l'une permettant de stocker la séquence binaire représentant la table de la machine T , sa configuration (état interne, caractère courant) et le troisième permettant de gérer les données (les entrées et les sorties).

Le fonctionnement de la machine universelle peut être schématisé comme suit:

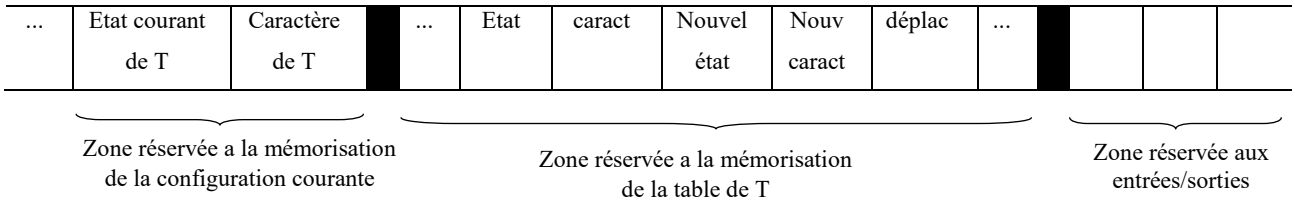


Figure 4.1 Schéma de fonctionnement de machine universelle

Algorithme Fonctionnement de la machine universelle U

Début : "M" et "ω" sur le premier ruban. **1ère étape** : "ω" sur le premier ruban

"M" sur le second ruban

"s" sur le troisième ruban

2ème étape :

Recherche sur le 2ème ruban de la transition de M

// lettre courante : position de la tête sur le 1er ruban

Si il existe une transition

Exécution de la transition :

- manipulation du 1er ruban

→ changer le caractère courant

→ déplacer la tête vers la droite ou vers la gauche

- écriture du nouvel état sur le 3ème ruban

Sinon arrêt.

4.3 Propriété des machines de Turing Universelles

L'universalité des machines de Turing se modélise sous deux formes différentes:

a. Universalité Externe : Toute modification du modèle ne change pas la classe des fonctions calculables par la machine de Turing. On peut citer:

- La tête de lecture peut en une seule étape écrire un nouveau symbole et se déplacer.

- La machine peut travailler sur plusieurs bandes en même temps.

- La machine non-déterministe est équivalente à une machine déterministe. On peut utiliser une machine à plusieurs bandes avec une tête de lecture qui peut écrire et se déplacer de manière atomique, le tout dirigé par un programme non déterministe.

b. Universalité interne : une seule machine peut simuler toutes les autres machines. Il s'agit de coder la machine de Turing en un nombre. L'avantage est que cela permet ensuite une manipulation plus abstraite de la notion de programme que de considérer des ensembles de quadruplets. Ce qui est équivalent au théorème de machine de Turing universelle suivant:

Théorème *(Machine de Turing universelle).*

$\exists u \in N, \ \forall x, y \in N^2$ *on a* $\varphi_u(x, y) = \varphi_x(y)$

u est le code d'une machine de Turing universelle

4.4 Problèmes de décision - Décidabilité

Dans un problème de décision, on a une propriété qui est soit vraie soit fausse pour chaque instance. L'objectif est de distinguer les instances positives E^+ (où la propriété est vraie) des instances négatives $E \backslash E^+$ (où la propriété est fausse).

4.4.1 Définitions

Définition 1. Un problème de décision P est la donnée d'un ensemble E, que l'on appelle l'ensemble des instances, et d'un sous-ensemble EÅ de E, que l'on appelle l'ensemble des instances positives.

Définition 2. *Un langage* $L \subset M^*$ *est dit* décidable *s'il est décidé par une machine de Turing.* Un langage qui n'est pas décidable est dit *indécidable*. On note D pour la classe des langages et des problèmes *décidables*.

Notons que la notion de décidabilité est rapprochée de celle de calculabilité, ce qui explique l'emploi des machines de Turing pour apporter une réponse au problème de la décision.

Exemple: *On discute la décidabilité du langage A constitué de la seule chaine s tel que:*

$$s = \begin{cases} 0 \; si \; l'etudiant \; n'a \; pas \; compris \\ 1 \;\; si \;\; l'étudiant \;\; a \;\; compris \end{cases}$$

***A* est décidable** par une machine de Turing car :

- Si *s* vaut 0 alors *A* est reconnu par la machine de Turing qui compare la lettre en face de la tête de lecture à 0 et accepte si elle vaut 0, et rejette sinon.
- Si *s* vaut 1 alors *A* est reconnu par la machine de Turing qui compare la lettre en face de la tête de lecture à 1 et accepte si elle vaut 1, et rejette sinon.

On peut alors énoncé qu'un problème de décision est dit décidable s'il existe un algorithme, une procédure mécanique qui termine en un nombre fini d'étapes, qui le décide, c'est-à-dire qui réponde par oui ou par non à la question posée par le problème. S'il n'existe pas de tels algorithmes, le problème est dit indécidable.

4.4.2 Indécidabilité. Problème de l'arrêt.

Un problème non décidable est un énoncé tel qu'il n'existe pas de preuve, ni de cet énoncé ni de sa négation, à partir des axiomes de la théorie.

Exemple; En théorie de la calculabilité, le **problème de l'arrêt** est un problème **indécidable**. Il consiste, étant donné un programme informatique quelconque (au sens machine de Turing), à dire s'il finira par s'arrêter ou non. Ce problème **n'est pas décidable**, car il n'existe pas de programme informatique qui prendrait comme entrée un autre programme informatique quelconque et qui ressortirait VRAI si le programme s'arrête et FAUX sinon.

Il existe dans la littérature, d'autre problèmes indécidable tels la question de savoir si oui ou non un énoncé de la logique du premier ordre est universellement valide (démontrable dans toute théorie). Aussi, le problème de la réduction qui est au cœur de nombreuses démonstrations d'indécidabilité et le problème de la correspondance . Le lecteur pourra se documenter sur ces derniers problèmes pour mieux cerner le problème de décidabilité.

Conclusion

Alan Turing a défini les principes des "Machine de Turing" afin d'étudier les limites de la calculabilité : ce qui est faisable par un ordinateur, ce qui ne le sera jamais quel que soit l'ordinateur ou le langage de programmation utilisé.

Une machine qui peut simuler n'importe quelle machine sur n'importe quelle entrée, est appelée une machine de Turing universelle U. Pour cela, la table de transition i.e programme d'une machine est donnée à la machine universelle. Ce programme codé sur le ruban devient une donnée de la machine universelle. La machine universelle est donc un modèle abstrait d'ordinateur, autrement dit elle décrit son fonctionnement. Dans ce dernier chapitre, la machine de Turing universelle a été présentée. Le concept de décidabilité d'un problème a été aussi abordé. Ce chapitre est clôturé par des exemples de problèmes indécidables.

Séries d'exercices Corrigés
par chapitre

Série d'exercice N°1

Chapitre 1. Introduction

Exercice 1.1

Soient i, n entiers positifs et a nombre réel positif. Montrer par induction les formules suivantes:

1. $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$
2. $\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$

Exercice 1.2

Prouver par induction que pour tout x et tout entier n

$$(-x)^n = \begin{cases} x^n & si\ n\ pair, \\ -x^n & sinon. \end{cases}$$

Exercice 1.3

Donner une définition inductive pour les cas suivants :

- Entiers Naturels N
- Entiers pairs

Exercice 1.4

Définir inductivement l'ensemble des expressions entièrement paranthesées formées à partir d'identificateurs pris dans un ensemble A et des opérateurs + et x.

Exercice 1.5

Utiliser le principe d'induction pour vérifier l'énoncé suivant pour tout n≥1 :

P_n = '' 7 devise $2^{n+2}+3^{2n+1}$''

Exercice 1.6

Démontrer par récurrence que pour tout entier n≥1, on a :

$$S_n = \sum_{k=1}^{n} k^3 = 1^3 + 2^3 + \cdots + n^3 = \frac{n^2(n+1)^2}{4}$$

Série d'exercice N°2

Chapitre 2 : Démonstration

Partie1: Logique Propositionnelle

Exercice 2.1

Soit l'ensemble des variables propositionnelle {p,f,s,l} suivant :

p: Il pleut

f : il fait froid

s : Meriem va sortir

l : Meriem porte son parapluie

Ecrire les phrases suivantes en formule propositionnelle :

1. S'il pleut, Meriem ne sort pas
2. S'il pleut, Meriem sort s'il ne fat froid
3. Si Meriem porte son parapluie, elle sort même s'il pleut
4. S'il ne pleut pas, Meriem ne portera pas son parapluie
5. Meriem porte son parapluie si et seulement si il pleut

Exercice 2.2

Trois collègues, Ali, Bachir et Chawki déjeunent ensemble chaque jour. Les affirmations suivantes sont vraies :

a. Si Ali commande un dessert, Bachir en commande un aussi.

b. Chaque jour, soit Bachir, soit Chawki, mais pas les deux, commandent un dessert.

c. Ali ou Chawki, ou les deux, commandent chaque jour un dessert.

d. Si Chawki commande un dessert, Ali fait de même.

Questions

1. Exprimer les données du problème comme des formules propositionnelles.
2. Peut-on déduire des modèles pour toute les formules propositionnelles ?

Exercice 2.3

Pour chacune des formules suivantes, donner sa représentation sous forme d'arbre:

1. $(P \Rightarrow Q \Rightarrow R) \Rightarrow (P \wedge Q \Rightarrow R)$
2. $(P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)$

Exercice 2.4

Dites si les formules suivantes sont satisfaisantes. Si oui, donner leurs modèles.

1. $p \rightarrow p$
2. $(p \rightarrow (q \rightarrow r)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r))$
3. $(\neg p \rightarrow q) \wedge (q \leftrightarrow r)$
4. $p \wedge \neg p$

Exercice 2.5

Relier les propositions équivalentes :

1. $\neg(p \wedge q)$	a. $(\neg p \wedge \neg q)$
2. $\neg(p \vee q)$	b. $q \rightarrow (\neg p)$
3. $p \rightarrow (\neg q)$	c. $(\neg p \vee \neg q)$
4. $\neg(p \rightarrow q)$	d. $p \wedge (\neg q)$

Exercice 2.6

Soient les formules suivantes :

$\alpha_1 : (x \vee y) \rightarrow (x \wedge y)$

$\alpha_2 : (x \wedge z) \rightarrow (x \rightarrow y)$

$\alpha : (x \vee z) \rightarrow (x \rightarrow y)$

Montrer que $\alpha 1, \alpha 2 \vDash \alpha$

Exercice 2.7

a. Mettre sous Forme Normale Conjonctive FNC

   1. $\neg p \rightarrow (q \wedge r)$
   2. $((b \vee c) \rightarrow a) \vee d$

b. Mettre les formules suivantes sous Forme Normale Disjonctive (FND) :

1. $p \wedge (p \rightarrow q)$
2. $(p \rightarrow \neg r) \wedge (\neg p \rightarrow \neg s)$

Série d'exercice N°2- bis

Chapitre 2 : Démonstration

Partie 2: Calcul des prédicats

Exercice 2.8

1. Ecrire les phrases suivantes en langage des prédicats LP1 :
   (a) Tous les plombiers sont des hommes ;
   (b) Pierre est riche ;
   (c) Si Pierre est un plombier, Pierre est riche ;
   (d) Quelques plombiers ne sont pas riches ;
   (e) Aucun plombier n'est riche.

   Utilisez les traductions suivantes pour les prédicats :

   P_x : x est plombier ;

   H_x : x est un homme ;

   R_x : x est riche.
2. Écrire à l'aide de quantificateurs les propositions suivantes :
   - Le carré de tout réel est positif.
   - Certains réels sont strictement supérieurs à leur carré.

Exercice 2.9

Déterminer les occurrences libres et liés dans les formules suivantes

a. $\forall z ((\forall x\ A(x,y) \rightarrow A(z,a))$

b. $\forall y\ A(z,y) \rightarrow \forall z\ A(z,y)$

c. $(\forall y\ \exists x\ \exists y\ B(x,y,f(x,y))) \vee \neg\ \forall x\ A(y,g(x))$

Exercice 2.10

a. Mettre les formules α et β sous la forme Prénexe et la formule ϕ sous forme Skolem

   1. $\alpha \equiv \forall x \forall y\ E(x,y) \rightarrow \exists z\ A(x,z)$
   2. $\beta \equiv \neg(\ p(x) \rightarrow ((\exists y\ q(x,y)) \wedge \exists y\ r(y))\ )$

b. Mettre la formule ϕ sous forme Skolem

$$\phi \equiv \exists u\ \forall x\ \exists y\ \forall z\ \exists t\ (\ p(x) \wedge q(y) \wedge r(x,z,t) \wedge s(y) \wedge k(u)\ )$$

Exercice 2.11

Mettre sous la forme Prenexe puis sous forme Skolem la formule F suivante:

$$F \equiv \forall x(\ p(x) \rightarrow \exists z\ (\neg \forall y\ (q(x, y) \rightarrow p(\ f(t)\ )\ ) \wedge \forall y(\ q(x, y) \rightarrow p(x)\ )\ )\ )$$

Exercice 2.12

A partir du système d'axiome du calcul propositionnel et de la règle du Modus Ponens, Montrer que l'on a :

1. $\dfrac{A}{B \rightarrow A}$
2. $\dfrac{(A \rightarrow B),(B \rightarrow C)}{(A \rightarrow C)}$
3. $\dfrac{A \rightarrow (B \rightarrow C)}{B \rightarrow (A \rightarrow C)}$

Exercice 2.13

Démontrer par déduction naturelle que :

a. $(\neg A \vee B) \rightarrow (A \rightarrow B)$
b. $(A \rightarrow B) \rightarrow (\neg A \vee B)$

Exercice 2.14

En utilisant la preuve pat déduction naturelle, démontrer que:

$(A \rightarrow B) \rightarrow [A \rightarrow (A \vee C)]$

Exercice 2.15

Par la méthode de résolution propositionnelle, établir la validité du raisonnement suivant:

" Si Lina gagne la confiance de ses partenaires et travaille dur alors elle aura une promotion. Si elle a une promotion alors elle achètera une nouvelle voiture. Elle n'a pas acheté une nouvelle voiture . Par conséquent, ou bien Lina n'a pas gagné la confiance de ses partenaires ou bien elle n'a pas travaillé dur."

Exercice 2.16

Soit la formule F suivante: $F \equiv (\neg q \rightarrow \neg p) \rightarrow (p \rightarrow q)$

Démontrer en utilisant la résolution par réfutation que F est une tautologie.

Exercice 2.17

Soient l'ensemble de formules $F = \{\forall x\ (p(x) \rightarrow q(x)), \forall x\ (q(x) \rightarrow r(x))\}$ et la formule $G = \forall x(p(x) \rightarrow r(x))$

Est ce que $F \models G$?

Série d'exercice N°3

Chapitre 3 . Modèles de calculs

Exercice 3.1

Développer une machine de Turing qui permet de calculer la fonction prédécesseur "Pred".

Exercice 3.2

Ecrire une machine de Turing qui rend le nombre 0 lorsque $x \neq 0$ sinon et rend 1 sinon

Exercice 3.3

Développer une machine de Turing qui permet de faire l'addition de deux nombre x et y qui se trouvant sur le ruban et séparé par une étoile.

Exercice 3.4

1. Expliquer le fonctionnement d'une machine de Turing à plusieurs rubans et plusieurs têtes.

Est ce que le modèle des machines de Turing à plusieurs rubans et plusieurs têtes est équivalent au modèle des machines de Turing à un ruban et une tête?

2. Comment simuler une Machine de Turing Multi-rubans avec une Machine de Turing Mono-ruban?

3. Discuter La simulation sur une machine Mono-ruban.

Série d'exercice N°4

Chapitre 4 . Calculabilité

Exercice 4.1

Compléter par une bonne réponse les assertions suivantes:

A1. Deux mt M1 et M2 sont équivalentes si et seulement si .....

A2. Un langage L est décidable si et seulement si .....

A3. Un langage L est indécidable si et seulement si .....

A4. Si m est le codage binaire de la mt M, ω un mot binaire et U est la machine de Turing universelle alors .....

A5.Le langage reconnu par la machine universelle U est .....

Exercice 4.2

Si une machine de Turing reconnaît un langage, existe t-il une machine de Turing qui reconnaît son complémentaire?

Exercice 4.3

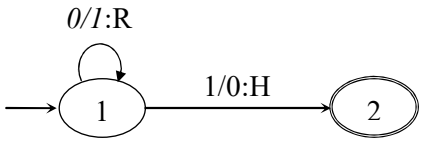
Montrer qu'une machine de Turing à $k > 1$ rubans de travail fonctionnant en temps t peut être simulée par une machine de Turing à un seul ruban de travail fonctionnant en temps $O(t^2)$.

Exercice 4.4

Discuter l'universalité d'une machine de Turing (Machine Universelle)

Exercice 4.5

Représenter en binaire la machine de Turing M suivante



Indication;

Pour coder cette représentation de M avec Σ= {0, 1} sur le ruban on utilisera un alphabet à 12 symboles $\Sigma_U = \Sigma \cup \{A,E,Q,(, :,),L,H,R, \$\}$ tel que:

- A, E, Q, (, :,) : Q pour les états quelconque par, A pour accepteur, E pour exception.

- L,H,R pour représenter les déplacements de la tête(gauche, halt et droit) lors des transitions.

- La séparation entre transitions est indiquée par la succession des symboles ")" et "(".

Corrigé de la série n°1

Chapitre 1. Introduction

Solution Exercice 1.1

1. Pour $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$

   Le cas de base est P(0), on obtient 0=0 donc vérifié.

   Pour prouver l'égalité par recursion on doit prouver $\sum_{i=0}^{n+1} i = \frac{(n+1)(n+2)}{2}$

   On a par hypothèse de recurrence:

   $\sum_{i=0}^{n+1} i = (n+1) + \sum_{i=0}^{n} i =$ (n+1)+ $\frac{(n+1)(n+2)}{2} = \frac{2(n+1)+n(n+1)}{2} = \frac{(n+1)(n+2)}{2}$ c

   cqfd

2. Pour $\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$

Pour le cas de base n=0 on trouve (1=1)

On procède par récurrence :

$\sum_{i=0}^{n+1} a^i = a^{n+1} + \sum_{i=0}^{n} a^i = a^{n+1} + \frac{a^{n+1}-1}{a-1} = \frac{(a-1)a^{n+1}+(a^{n+1}-1)}{a-1} = \frac{a^{n+2}-1}{a-1}$

cqfd

cqfd: Ce qui fallait démontrer

Solution Exercice 1.3

1. Une définition inductive des entiers naturels :

(B) 0 ∈ N;

(I) n ∈ N ⇒ n +1 ∈ N.

Ou encore : L'ensemble N défini par: $\frac{}{0 \in X}$ $\frac{n \in X}{n+1 \in X}$

2. Une définition inductive des entiers pairs :

(B) $0 \in P$;

(I) $n \in P \Rightarrow n+2 \in P$.

Exercice 1.5

Condition de base: $n \geq 1$ on prend n=1

(B) $P_1=2^{1+2}+3^{2x1+1}=8+27=35=5x7$ donc Vrai

Condition d'induction (I)

On suppose que l'énoncé Pn soit vrai pour une valeur de n: "7 divise $2^{n+2}+3^{2n+1}$".

Alors 7 divise $2^{n+2}+3^{2n+1}=7k$ (avec k entier).

On vérifie alors cette propriété pour (n+1) :

$$2^{(n+1)+2}+3^{2(n+1)+1}= 2^{(n+2)+1}+3^{2n+2+1}$$
$$= 2.2^{(n+2)}+3.3^{2n+1+2}$$
$$= 2.2^{(n+2)}+3^2.3^{2n+1}$$
$$= 2.2^{(n+2)}+9.3^{2n+1}$$
$$= 2.2^{(n+2)}+(2+7).3^{2n+1}$$
$$= 2.(2^{(n+2)}+3^{2n+1})+7.3^{2n+1}$$
$$= 2.7k+7.3^{2n+1} \quad (\text{car } 2^{n+2}+3^{2n+1}=7k)$$
$$=7.(2k+3^{2n+1})$$

Donc $2^{(n+1)+2}+3^{2(n+1)+1}=$ est un multiple de 7 c.-à-d. que 7 divise $2^{(n+1)+2}+3^{2(n+1)+1}$.

Corrigé de la série n°2

Chapitre 2 : Démonstration

Solution Exercice 2.2

1. On introduit des variables propositionnelles a, b et c qui représentent le fait que Ali(a), Bachir(b) et Chawki(c) prennent un dessert. On traduit ainsi le problème :

   (a) $a \Rightarrow b$

   (b) $(b \wedge \neg c) \vee (\neg b \wedge c)$

   (c) $a \vee c$

   (d) $c \Rightarrow a$

2. On peut faire une table de vérité pour trouver tous les modèles possibles :

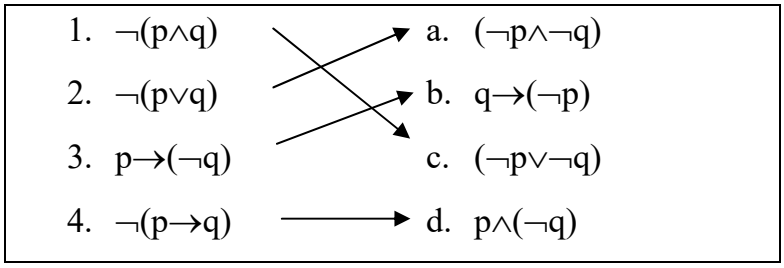
a	b	C	$a \Rightarrow b$	$(b \wedge \neg c) \vee (\neg b \wedge c)$	$a \vee c$	$c \Rightarrow a$
F	F	F	V	F	F	V
F	F	V	V	V	V	F
F	V	F	V	V	F	V
F	V	V	V	F	V	F
V	F	F	F	F	V	V
V	F	V	F	V	V	V
V	V	F	V	V	V	V
V	V	V	V	F	V	V

La seule interprétation qui rend vrai les quatre affirmations correspond avant derniere ligne (7) dans laquelle Ali et Bachir commandent un dessert mais pas Chawki.

Solution Exercice 2 4

1. Tautologie (toujours vrai)
2. Tautologie (toujours vrai)
3. Satisfaisante, Modeles (0,1,1) ; (1,0,0) ; (1,1,1)
4. Antilogie (toujours fausse)

Solution Exercice 2.5



1-c, 2-a, 3-b, 4-d

Solution Exercice 2.6

Utilisation de la table de vérité

Solution Exercice 2.8

1.

a) D = {les chats}, soit le prédicat G(x) : "x est gris". On aura la formule $\forall x G(x)$;

b) Tous les plombiers sont des hommes : $\forall x(Px)\ Hx)$

c) Pierre est riche . a : Pierre, R_a

d) Si Pierre est un plombier, Pierre est riche. a : Pierre, $P_a \rightarrow R_a$

e) Quelques plombiers sont riches. $\exists x\ (P_x \wedge R_x)$

f) Aucun plombier n'est riche. $\neg\exists(P_x \wedge R_x)$

2. a- $\forall x \in R, x^2 \geq 0$

b- $\exists x \in R / x \geq x^2$

Solution Exercice 2.9

Déterminer les occurrences libres et liés dans les formules suivantes

a. $\forall z ((\forall x A(x,y) \rightarrow A(z,a))$

- L'occurrence de x est liée.
- L'occurrence de y est libre.
- L'occurrence de z est liée.

b. $\forall y A(z,y) \rightarrow \forall z A(z,y)$

- La 1ière occurrence de y est liée, la seconde est libre
- La 1ière occurrence de z est libre, la seconde est liée.

c. $(\forall y \exists x \exists y B(x,y,f(x,y))) \vee \neg \forall x A(y,g(x))$

- Les occurrences de x sont liées.
- Les deux premières occurrences de y sont liées, la troisième est libre.

Solution Exercice 2.10

a. Soient les formules α et β suivantes: Mettre sous la forme Prenexe:

1. $\alpha \equiv \forall x \forall y E(x,y) \rightarrow \exists z A(x,z)$
2. $\beta \equiv \neg(p(x) \rightarrow ((\exists y q(x,y)) \wedge \exists y r(y)))$

1. Soit la formule $\alpha \equiv \forall x \forall y E(x,y) \rightarrow \exists z A(x,z)$

$\alpha \equiv \forall x \ \forall y E(x,y) \rightarrow \exists z A(x,z)$

$\alpha \equiv \forall t \ \forall y \ E(t,y) \rightarrow \exists z A(x,z)$

$\alpha \equiv \exists t \ \exists y \ (E(t,y) \rightarrow \exists z A(x,z))$

$\alpha \equiv \exists t \ \exists y \ \exists z (E(t,y) \rightarrow A(x,z))$

2. Soit la formule $\beta \equiv \neg(p(x) \rightarrow ((\exists y\, q(x,y)) \wedge \exists y\, r(y)))$:

Éliminer $\rightarrow$:

$$\beta \equiv \neg(\neg p(x) \vee ((\exists y\, q(x,y)) \wedge \exists y\, r(y)))$$

Rentrer $\neg$:

$$\beta \equiv \neg\neg p(x) \wedge \neg((\exists y\, q(x,y)) \wedge \exists y\, r(y))$$

$$\beta \equiv p(x) \wedge \neg((\exists y\, q(x,y)) \wedge \exists y\, r(y))$$

$$\beta \equiv p(x) \wedge (\neg(\exists y\, q(x,y)) \vee \neg\exists y\, r(y))$$

$$\beta \equiv p(x) \wedge ((\forall y\, \neg q(x,y)) \vee \neg\exists y\, r(y))$$

$$\beta \equiv p(x) \wedge ((\forall y\, \neg q(x,y)) \vee \forall y\, \neg r(y))$$

Renommer $\forall y\, \neg r(y)$ en $\forall z\, \neg r(z)$:

$$\beta \equiv p(x) \wedge ((\forall y\, \neg q(x,y)) \vee \forall z\, \neg r(z))$$

Sortir $\forall y$ et $\forall z$:

$$\beta \equiv p(x) \wedge \forall y\, (\neg q(x,y) \vee \forall z\, \neg r(z))$$

$$\beta \equiv \forall y\, (p(x) \wedge (\neg q(x,y) \vee \forall z\, \neg r(z)))$$

$$\beta \equiv \forall y\, (p(x) \wedge \forall z\, (q(x,y) \vee \neg r(z)))$$

$$\beta \equiv \forall y\, \forall z\, (p(x) \wedge (\neg q(x,y) \vee \neg r(z)))$$

b. Soit la formule $\phi \equiv \exists u\, \forall x\, \exists y\, \forall z\, \exists t\, (p(x) \wedge q(y) \wedge r(x,z,t) \wedge s(y) \wedge k(u))$. Mettre la formule ϕ sous forme Skolem

1. Remplacer u par la constante a :

$$\phi \equiv \forall x\, \exists y\, \forall z\, \exists t\, (p(x) \wedge q(y) \wedge r(x,z,t) \wedge s(y) \wedge k(a))$$

2. Remplacer y par la fonction f(x) :

$$\phi \equiv \forall x\, \forall z\, \exists t\, (p(f(x)) \wedge q(f(x)) \wedge r(f(x),z,t) \wedge s(y) \wedge k(a))$$

3. Remplacer t par la fonction g(x,z) :

$$\phi \equiv \forall x\, \forall z\, (p(f(x)) \wedge q(f(x)) \wedge r(f(x),z,g(x,z)) \wedge s(y) \wedge k(a))$$

Solution Exercice 2.12

1. $\frac{\mathbf{A}}{\mathbf{B}\rightarrow\mathbf{A}}$

   a. A hypothèse

   b. A→(B→A) // selon Axiome 1

   c. B→A // selon règle Modus Ponens (a et b)

2. $\frac{(\mathbf{A}\rightarrow\mathbf{B}),(\mathbf{B}\rightarrow\mathbf{C})}{(\mathbf{A}\rightarrow\mathbf{C})}$

   a. A→ B hypothèse

   b. B→ C hypothèse

   c. A→(B→ C) // selon (b et 1.)

   d. A→(B→ C) → (A→B)→(A→C) // selon Axiome 2

   e. (A→B)→(A→C) // selon règle Modus Ponens c et d

   f. (A→C) // selon règle Modus Ponens a et e

3. Même principe que 2

Solution exercice 2.13

a. (¬ A∨B) → (A→B)

1	¬ A∨B	, Hyp 1
2	A	, Hyp 2
3	¬ A∨B	, reit 1
4	B	, e ∨, 2,3
5	A→B	, i →,2,4

b. Même principe que dans a.

Solution Exercice 2.15

Soient les variables propositionnelles:

Lina gagne la confiance de ses partenaires et travaille dur alors elle aura une promotion. Si elle a une promotion alors elle achètera une nouvelle voiture. Elle n'a pas acheté une nouvelle voiture . Par conséquent, ou bien Lina n'a pas gagné la confiance de ses partenaires ou bien elle n'a pas travaillé dur

c=" Lina gagne la confiance de ses partenaires "

d=" Lina travaille dur "

p="Lina a une promotion"

v="Lina achète une nouvelle voiture "

Le formalisme du raisonnement en utilisant la résolution par réfutation:

$(c \wedge d) \rightarrow p,\ p \rightarrow v,\ \neg v,\ (c \wedge d) \models \square$

$\neg c \vee \neg d \vee p,\ \neg p \vee v,\ \neg v,\ c, d \models \square$

On a donc :

C1= $\neg c \vee \neg d \vee p$

C2= $\neg p \vee v$

C3= $\neg v$

C4= c

C5= d

C6 = $\neg p$ Res (C2,C3)

C7= $\neg c \vee \neg d$ Res (C1,C6)

C8= $\neg d$ Res(C4, C7)

C9= $\square$ Res (C5, C8)

Ainsi, On déduit que le raisonnement donné est valide.

Solution Exercice 3.5

Soit la formule F suivante: $F \equiv (\neg q \rightarrow \neg p) \rightarrow (p \rightarrow q)$

Démontrons en utilisant la résolution par réfutation que F est une tautologie

$\neg F \equiv \neg ((\neg q \rightarrow \neg p) \rightarrow (p \rightarrow q))$

$\neg F \equiv \neg ((\neg(\neg q \rightarrow \neg p) \vee (p \rightarrow q))$

$\neg F \equiv (q \vee \neg p) \wedge (p \wedge \neg q)$

L'application de la résolution:

$C_1 = q \vee \neg p$

$C_2 = p$

$C_3 = \neg q$

$C_4 = q \quad Res(C_1, C_2)$

$C_5 = \square \quad Res(C_3, C_4)$

Donc $\neg F$ est une contradiction et F est une tautologie

Corrigé de la série n°3

Chapitre 3 . Modèles de calculs

Solution Exercice 3.2

Ecrire une machine de Turing qui rend le nombre 0 lorsque x≠0 sinon et rend 1 sinon

$$q_0 \underline{0} \rightarrow q_f \underline{1}$$

$$\text{ou} \quad q_0 \underline{x} \rightarrow q_f \underline{0}$$

- $q_0 1 D q_1$
- $q_1 0 1 q_2$
- $q_2 1 G q_f$
- $q_1 1 0 q_2$
- $q_2 0 D q_3$
- $q_3 1 0 q_2$
- $q_3 0 G q_4$
- $q_4 0 G q_4$
- $q_4 1 1 q_f$

Solution Exercice 3.4

1. En fonction de l'état et du symbole lu sur chacun des rubans, la machine peut

- changer d'état,
- écrire un symbole sur chaque ruban,
- déplacer chaque tête vers la droite ou la gauche indépendamment les unes des autres

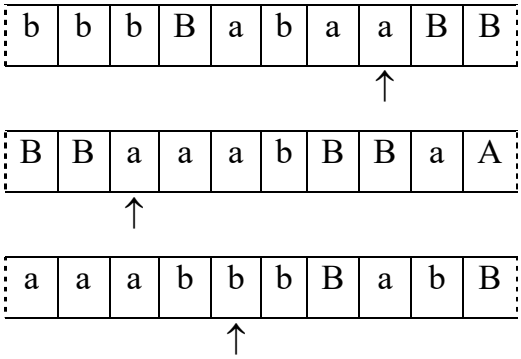
Oui le modèle des machines de Turing à plusieurs rubans et plusieurs têtes est équivalent au modèle des machines de Turing à un ruban et une tête.

2. L'idée est de simuler une machine à k rubans et k têtes sur un alphabet Γ dont le symbole blanc est B, avec une machine mono-ruban sur l'alphabet $\Gamma' = (\Gamma \times \{B, \#\})^k \cup \Sigma$ dont le symbole blanc est (B, B, B, B, B, B).

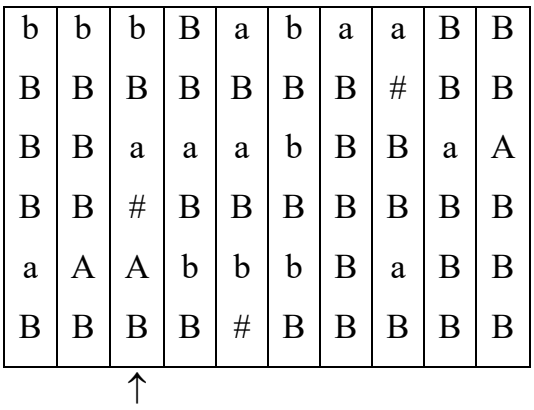
L'ajout de Σ sert à ce que la machine mono-ruban puisse être lancée sur les mêmes entrées que la machine multi-ruban, afin de reconnaître le même langage. On peut voir un symbole de la machine mono-ruban comme 2k sous-symboles (k couples de Γ × {B, #}).

Chacun des k rubans avec sa tête sera représenté par deux sous-symboles : le premier correspond au contenu de ce ruban et sur le second nous avons un # à l'endroit où se trouve la tête de lecture de ce ruban (B partout ailleurs).

- Machine à trois rubans dans l'état q:



- Machine Mono-Ruban dans l'état q:



3. L'exécution sur la machine Mono-ruban (qui simule la machine à k rubans) consiste à toujours ramener l'unique tête de lecture/écriture de la machine sur la case la plus à gauche de la machine Multi-ruban simulée. Alors pour simuler une étape de la machine à k-rubans, la machine Mono-ruban fait un aller-retour jusqu'à la tête la plus à droite :

- sur l'"aller" la machine Mono-ruban se souvient dans son état des sous-symboles sous les têtes de chaque ruban simulé, une fois arrivée à la fin de l'"aller", la machine Mono-ruban sait donc quelle transition de la machine à k rubans elle doit simuler,

- sur le"retour" la machine Mono-ruban met à jour tous les sous-symboles de ruban et tous les sous-symboles de têtes selon la transition à simuler, dans l'ordre où elle les rencontre, en s'arrêtant sur la tête la plus à gauche (c-ad $k^{ième}$ sous-symbole # trouvé).

Corrigé de la série n°4

Chapitre 4 . Calculabilité

Solution Exercice 4.2

Faux. et on peut donner un contre-exemple :

La machine de Turing universelle U reconnaît le langage universel

$L_U = \{(m, \omega) \mid U(m, \omega) = V\} = L(U)$.

Or $\overline{LU}$ n'est pas reconnaissable par une Machine de Turing M.

Les éléments de LU sont les couples (m, ω) que la machine universelle U n'accepte pas.

$\overline{L_U} = (m, \omega) \in M \times \{0, 1\}^* \mid U(m)(\omega) \neq \text{Accepté}\}$

Solution Exercice 4.3

On utilise le principe décrit pour la machine universelle U et qui consiste à copier le contenu des k rubans de travail "colonne par colonne".

La nouvelle machine doit alors parcourir tout son ruban de travail pour simuler une étape, d'où l'augmentation quadratique du temps.

Solution Exercice 4.4

Une machine Mu est appelée une machine de Turing universelle si elle peut simuler n'importe quelle machine sur n'importe quelle entrée (si on lui donne une description de la machine à simuler).

Solution Exercice 4.5

MT:

(Q : 1)	$	(Q : 1)	0	1	R	(Q : 1)	(Q : 1)	1	0	H	(A : 01)

Le codage binaire des machine de Turing) noté M = {m ∈ {0,1}* tel que m = $[M]_2$, M∈ MT} est l'ensemble des mots binaires qui correspondent à des MT sur l' alphabet {0,1}.

Codage binaire de Σ_U

- 0 : 0000
- 1 : 1111
- Q : 1110
- A : 0111
- E : 1101
- $: 0001
- L : 1100
- H : 0110
- R : 0011
- (: 1000
-) : 0010
- : : 0100

$m = [M]_2$ = 1000 1110 0100 1111 0010 0001 1000 1110
0100 1111 0010 0000 1111 0011 1000 1110
0100 1111 0010 1000 1110 0100 1111 0010
1111 0000 0110 1000 0111 0100 0000 1111
0010

Enoncé du TP
& Corrigé Type

TP

Satisfiabilité d'une formule

Enoncé

Soit une φ une formule propositionnelle. φ est en forme normale conjonctive (FNC) si elle est une conjonction de clauses. Une *clause* étant une disjonction de littéraux (formule atomique). On dit que φ est *satisfiable* (satisfaisante) si et seulement s'il existe une interprétation I telle que I(φ) = 1. Sinon φ est insatisfiable ou contradictoire.

Problème (SAT)

Entrée : une formule propositionnelle φ

Question : φ est-elle satisfiable ?

Sortie : Si oui afficher modèle

Objectif

Le but de ce TP est d'implémenter un algorithme qui permet de décider si une formule φ de logique propositionnelle en forme normale conjonctive (FNC) est satisfiable ou non.

Travail Demandé

- Ecrire une méthode satisfiable qui construit successivement toutes les interprétations possibles des propositions, et retourne vrai dès qu'elle trouve un modèle.
- Retrouver les modèles pour la Satisfiabilité d'une formule φ sinon dire que φ est non satisfiable.

Langage de programmation :

Prolog (de préférence), OCaml, Java, …

Travail en Binômes - A remettre (CD + Rapport détaillé)

Corrigé Type du TP

Satisfiabilité d'une formule

L'un des problèmes les plus importants de l'informatique (et aussi des mathématiques) est le problème SAT, pour SATisfaisabilité (ou SATisfiabilité). Ce problème capture la difficulté de toute une famille de problèmes difficiles que l'on rencontre dans un très grand nombre de problèmes théoriques et pratiques au sens de la théorie de la complexité.

Le problème SAT s'exprime en logique propositionnelle. Son apparente simplicité permet en effet l'élaboration d'algorithmes compacts et efficaces en pratique pour attaquer toute une classe de problèmes importants, mais intraitables en théorie.

En fait, une logique se définit d'abord de manière syntaxique. Une fois que la sémantique est fixée, on peut se demander si, étant donnée une formule f , il existe une interprétation I qui la satisfait (on aurait $||f||_I = V$). Si c'est le cas, on dit que I est un modèle de f. Inversement, si $||f||_I=F$, on dit que I falsifie f et que I est un contre-modèle de f.

Le problème étudié dans ce TP est donc de savoir si une formule donnée admet un modèle SAT. Ce dernier est en fait le premier problème démontré comme NP-complet: il peut être résolu en temps polynomial par une machine non-déterministe. Autrement dit, on sait déterminer en temps polynomial si une "solution candidate" (ici une interprétation I) permet d'affirmer que la formule est satisfiable.

Algorithmes de résolution de SAT

De nombreux algorithmes ont été proposés pour résoudre le problème SAT. Nous présentons deux qui peuvent être implémenter avec plusieurs langages de programmation (Python, Java, Prolog). On note que la formule dont il faut décider la satisfaisabilité doit être en Forme Normale Conjonctive (FNC).

a. Algorithme de Quine

La méthode de Quine utilise des formules mises au préalable sous forme normale conjonctive, assimilées donc à des ensembles de clauses. Le principe de cette méthode est de parcourir l'arbre de toutes les solutions (l'arbre dont les branches complètes sont les valuations, appelé arbre sémantique). Chaque fois qu'une variable est affectée à la valeur x $\in \{0,1\}$, le calcul se fait récursivement avec $C[p \leftarrow \bot]$, si x = 0, et avec $C[p \leftarrow T]$, sinon.

Algorithme de Quine

Entrée : un ensemble de clauses C

Sortie : vrai si C est satisfaisable ou f aux sinon

Simplifier l'ensemble de clauses;

Si C = ∅ retourner vrai

Si C contient la clause ⊥ Retourner faux

Choisir le prochain p ∈ Prop apparaissant dans une clause

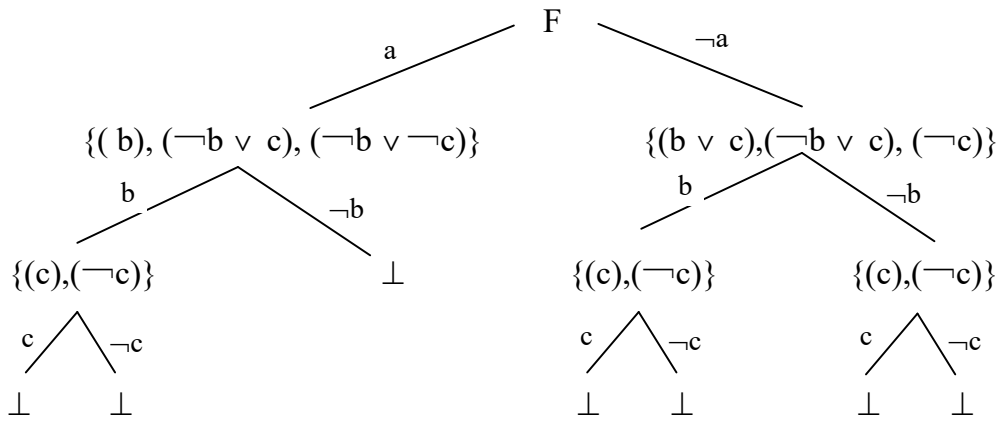
Si Quine(C[p ← ⊥]) = vrai alors retourner vrai

Sinon retourner Quine(C[p ← T])

Exemple: Soient un ensemble des variables {a, b, c}et la F formule constituée d'ensemble de clauses telle que: F = {(a ∨ b ∨ c), (¬a ∨ b), (¬b ∨ c), (¬c ∨ a), (¬a ∨ ¬b ∨ ¬c)}

.

L' arbre sémantique construit par la méthode Quine peut être représenter comme suit:



2. Algorithme DPLL

L'algorithme DPLL , du nom de ses auteurs Davis, Putnam, Logemann et Loveland, est considéré, à ce jour, comme l'une des méthodes les plus efficaces parmi celles permettant de résoudre le problème SAT ; la plupart des outils de satisfaction de contraintes (« SAT solvers », en anglais) implémentent une variante de cet algorithme. Il peut être vu comme un raffinement de la méthode de Quine ; l'amélioration principale qu'il apporte est:

- La réduction du branchement via la propagation des clauses unitaires ;
- L'utilisation d'heuristiques pour accélérer le parcours des solutions.

La propagation unitaire est un moyen de prendre en compte l'affectation d'une variable à une valeur selon les deux principes suivants :

1. Eliminer des clauses les littéraux faux vis-à-vis de l'affectation courante des valeurs aux variables;

2. Pour les clauses unitaires, c'est-à-dire les clauses possédant un seul littéral, forcer l'affectation de la variable de façon à rendre la clause vraie.

L'algorithme DPLL va appliquer la propagation unitaire à chaque nœud de l'arbre de recherche. De plus, si certaines variables se voient affectées une valeur lors de ce processus, la propagation unitaire leur est également appliquée, jusqu'à ce que plus aucune variable ne se voit affectée de valeur par ce biais. L'algorithme DPLL est présenté ci-dessous.

Algorithme DPLL

Fonction DPLL(A) :

```
Appliquer la propagation unitaire, de manière itérative
    jusqu'à obtention d'un point fixe
Si A ne contient que des clauses satisfaites
    renvoyer SAT
Si A contient la clause vide
    renvoyer UNSAT
Choisir une variable I
Renvoyer DPLL(A ∪ {I}) ∨ DPLL(A ∪ {¬I})
```

De nombreuses améliorations peuvent être apportées à la procédure DPLL, Elles concernent généralement un ou plusieurs points suivants :

1. Heuristique de branchement : comment sélectionner la prochaine variable à affecter ?
2. Simplification de la formule : comment simplifier la formule ?
3. Traitement des échecs : que doit-on faire en cas de conflits ?

L'étudiant pourra ainsi proposer une modification de l'algorithme DPLL en retrouvant les réponses adéquates aux questions précédentes.

Examen Finale & Corrigé Type

Examen de Rattrapage

Département d'informatique Documents non autorisés
1MSIQ / Module : LFI1/ 2021
CC Dr Benhamza

Examen Final

Exercice 1:

En utilisant le principe de la récursivité, définir inductivement les expressions arithmétiques des formules bien formée sur l'alphabet Σ_{exp}={0,1,2..9,+,-,*,/,(,)}

Exercice 2

Soit l'énoncé suivant: " Si Nazim gagne la confiance de ses dirigeants et travaille dur alors il aura une promotion. S'il a une promotion alors il achète une voiture. Nazim n'a pas acheté une voiture. Par conséquent , Nazim n'a pas gagné la confiance de ses dirigeants ou bien il n'a pas travaillé dur."

Etablir la validité de l'énoncé en utilisant la méthode de résolution.

Exercice 3

1. En utilisant la démonstration a la Hilbert-Frege, déduire la règle de permutation suivante :

$$\frac{A \rightarrow (B \rightarrow C)}{B \rightarrow (A \rightarrow C)}$$

A1: $(X_1 \Rightarrow (X_2 \Rightarrow X_1))$ (axiome 1 pour l'implication) ;

A2: $((X_1 \Rightarrow (X_2 \Rightarrow X_3)) \Rightarrow ((X_1 \Rightarrow X_2) \Rightarrow (X_1 \Rightarrow X_3)))$ (axiome 2 pour l'implication) ;

A3: $(X_1 \Rightarrow \neg\neg X_1)$ (axiome 1 pour la négation) ;

2. Démontrer par déduction naturelle que $(A \rightarrow B) \leftrightarrow (\neg B \rightarrow \neg A)$

3. Soit l'énonce suivant:

Les hiboux sont des oiseaux. La nuit tous les oiseaux peuvent voler.

En utilisant la résolution par réfutation, montrer que "La nuit tous les hiboux peuvent voler".

Indication: utiliser les prédicats suivants:

- h(x) : x est un hiboux
- o(x): x est un oiseau
- v(x): x peut voler la nuit

Exercice 4

Développer une machine de Turing qui permet de calculer la fonction prédécesseur (Pred) d'un entier définie par:

$$\text{Pred} = \begin{cases} x - 1 & si\ x > 0 \\ 0 & si\ x = 0 \end{cases}$$

Bon travail

Département d'informatique Documents non autorisés
1MSIQ / Module : LFI1/ 2021
CC Dr Benhamza

Corrigé Type Examen Final

Solution Exercice 1

Définition d'un Nombre en base 10 sans débuter par 0 (0023 non valide)

(B) $a \in N$ pour $a \in \{1,2,\ldots 9\}$;

(I) $g \in N \Rightarrow ga \in N$ pour $a \in \{0,1,2,\ldots 9\}$

On définit ensuite les expressions arithmétiques est la partie de Σ^*_{exp} définie inductivement par:

(B) $0 \in Arith$;

(B) $N \subset Arith$;

(I) $g, d \in Arith \Rightarrow g + d \in Arith$;

(I) $g, d \in Arith \Rightarrow g * d \in Arith$;

(I) $g, d \in Arith \Rightarrow g / d \in Arith$;

(I) $g, d \in Arith \Rightarrow g - d \in Arith$;

(I) $g \in Arith \Rightarrow (g) \in Arith$;

Exemple : $(1+2*4*(3+2)) \in Arith$ est une expression valide.

Solution Exercice 2

Soit l'énoncé suivant: " Si Nazim gagne la confiance de ses dirigeants et travaille dur alors il aura une promotion. S'il a une promotion alors il achète une voiture. Nazim n'a pas acheté une voiture. Par conséquent , Nazim n'a pas gagné la confiance de ses dirigeants ou bien il n'a pas travaillé dur."

Pour établir la validité de l'énoncé on utilise la méthode de résolution.

Soient les variables propositionnelles

c: Nazim gagne la confiance de ses dirigeants

d: Nazim travaille dur

p: Nazim a une promotion

v: Nazim achète une nouvelle voiture

$(c \wedge d) \rightarrow p,\ p \rightarrow v,\ \neg v \qquad \neg c \vee \neg d$

équivalent à: $\neg c \vee \neg d \vee p,\ \neg p \vee v,\ \neg v\ ,\ c,\ d \models \square$

On a:

$C_1 = \neg c \vee \neg d \vee p$

$C_2 = \neg p \vee v$

$C_3 = \neg v$

$C_4 = c$

$C_5 = d$

$C_6 = \neg p$ Res(C_2, C_3)

$C_7 = \neg c \vee \neg d$ Res(C_1, C_6)

$C_8 = \neg d$ Res(C_4, C_7)

$C_6 = \square$ Res(C_5, C_8)

On deduit que le raisonnement de l'énoncé est valide

Solution Exercice 3

1. En utilisant la démonstration a la Hilbert-Frege, on va déduire la règle de permutation suivante :

$$\frac{A \rightarrow (B \rightarrow C)}{B \rightarrow (A \rightarrow C)}$$

1. $A \rightarrow (B \rightarrow C)$ Hypothèse
2. $A \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow (A \rightarrow C)$ Axiome 2
3. $(A \rightarrow B) \rightarrow (A \rightarrow C)$ Règle Modus Ponens (1 et 2)
4. $B \rightarrow (A \rightarrow B)$ Axiome 1
5. $B \rightarrow (A \rightarrow C)$ par transitivité 3 et 4

2. Démontrer par déduction naturelle que $(A \rightarrow B) \leftrightarrow (\neg B \rightarrow \neg A)$

Démonstration présentée pendant le cours

3. Soit l'énonce suivant:

Les hiboux sont des oiseaux. La nuit tous les oiseaux peuvent voler.

En utilisant la résolution par réfutation, montrer que "La nuit tous les hiboux peuvent voler".

.Pour formaliser cet énonce, on utilise les prédicats suivants:

- h(x) : x est un hiboux
- o(x): x est un oiseau
- v(x): x peut voler la nuit

On met d'abord le système sous forme Prenexe, puis sous forme de Skolem et enfin sous forme Clausale.

• Forme Prenexe:

$A_1 : \forall x\ h(x) \Rightarrow o(x) \equiv \forall x \neg h(x) \vee o(x)$

$A_2 : \forall x\ o(x) \Rightarrow v(x) \equiv \forall x \neg o(x) \vee v(x)$

$\neg A_3 : \neg(\forall x\ h(x) \Rightarrow v(x)) \equiv \exists x\ h(x) \wedge \neg v(x)$

• Forme de Skolem:

$A_1 : \neg h(x) \vee o(x)$

$A_2 : \neg o(x) \vee v(x)$

$\neg\ A_3 : h(a) \wedge \neg v(a)$

• **Forme Clausale**

C1 : ¬h(x) ∨ o(x)

C2 : ¬o(x) ∨ v(x)

C3 : h(a)

C4 : ¬v(a)

• **Résolution**

C1,C3 |= C4 avec l'unification {x|a} et C5 ≡ o(a)

C5,C2 |= C6 avec l'unification {x|a} et C5 ≡ v(a)

C6, C4 |= □ avec l'unification {x|a}

Solution Exercice 4

Développement d'une machine de Turing qui permet de calculer la fonction prédécesseur (Pred) d'un entier définie par:

$$\text{Pred} = \begin{cases} x - 1 & si\ x > 0 \\ 0 & si\ x = 0 \end{cases}$$

- q_0 1 D q_1
- q_1 1 D q_2 q_1 0 G q_f **cas x=0**
- q_2 1 D q_2
- q_2 0 G q_3
- q_3 1 0 q_4
- q_4 0 G q_5
- q_5 1 G q_5
- q_5 0 D q_f

Département d'informatique
1Master SI /2021
CC Dr Benhamza

Module : Logique et Fondements de l'informatique 1

Durée:1h30

Examen de Rattrapage

Exercice 1

Modéliser en logique des prédicats les assertions suivantes

1. Tous les étudiants, sauf Nazim, sont présents
2. Aucun enfant ne fait jamais aucune bêtise
3. Tout les étudiants ont lu un livre de logique

Exercice 2

Utilisez la méthode de la résolution (par réfutation) pour prouver que la formule ci-dessous est un théorème.

$$(\neg A \rightarrow \neg B) \rightarrow (B \rightarrow A)$$

Exercice 3

Mettre les formule suivantes sous forme de Prenexe et sous forme Skolem

1. $\forall x\ (p(x) \rightarrow \exists y\ q(x,y))$
2. $\forall x\ ((\exists z\ p(x,z) \wedge p(y,z)) \rightarrow \forall y\ \exists u\ q(x,y,u))$

Exercice 4

Ecrire une machine de Turing qui rend le nombre 0 lorsque $x \neq 0$ sinon et rend 1 sinon

Bon travail

Conclusion Générale

Conclusion Générale

Cet polycopié de cours couvre les principes des raisonnements logique et par induction. On a essayer de faire découvrir aussi les aspects fondamentaux de la décidabilité et de la calculabilité qui sont des faces importante de l'informatique, plus précisément de la théorie de calcul (Theory of computation). Cette théorie s'intéresse à une question essentielle: quels problèmes peut-on (*pratiquement*) résoudre à l'aide d'un ordinateur ?

Le terme de " DÉCIDABILITÉ " est relatif aux problèmes pouvant être exprimés dans le formalisme des mathématiques ou de l'informatique. Il faudra donc trouver une traduction du problème dans un langage accessible au raisonnement automatique (raisonnement par induction, raisonnement logique). La catégorie de problèmes exprimés sous forme de conjecture fait partie de ce que l'on nomme les problèmes de décision.

Le terme de "CALCULABILITÉ" s'intéressèrent lui à la définition d'un processus effectif de calcul (se réfère à un algorithme ou même un ordinateur). En fait, la calculabilité cherche à identifier la classe des fonctions qui peuvent être calculées à l'aide d'un algorithme et à appliquer ces concepts à des questions fondamentales des mathématiques. Ceci permet d'appréhender ce qui est calculable et de ce qui ne l'est pas et permet de voir les limites des problèmes qui peuvent être résolus par les ordinateurs.

Ce document comporte quatre chapitre:

Le chapitre 1 est consacré au raisonnement mathématique par récurrence on y aborde notamment le théorème du point fixe formalisé pour l'ensemble des fonctions inductives

Le chapitre 2 est consacré à théorie de la démonstration ou de preuve. Un système de preuve est la donne d'un ensemble de formules appelées axiomes, et d'un nombre fini de règles. On a présenté trois stratégies de démonstration: A la Hilbert-Frege, Par Déduction Naturelle et Preuve par Résolution. Par ailleurs, on a jugé nécessaire de faire un rappel des principes de la logique propositionnelle et du calcul des prédicats dans la partie introduction. Les notions de validité, de modèles et de formes normales sont aussi exposées.

Le chapitre 3 présente les modèle de calculs et la théorie de la calculabilité qui a été développée par le mathématicien David Hilbert et qui cherche à répondre à la question "Qu'est-ce qui est calculable ?". La modélisation du processus du calcul via des modèles formels et qui permettent d'appréhender les limites non-physiques mais bien conceptuelles du calcul et de l'informatique en général. Ils modélisent formellement la notion de calcul effectif ou encore celle d'algorithme. La résolution d'un problème à l'aide d'un processus effectif, consiste en des manipulations d'un ensemble fini de symboles selon un ensemble fini de règles non ambiguës. La Machine de Turing est présenté pour valider ce processus avec des exemples applicatifs.

Le chapitre 4 aborde la machine de Turing universelle pour appuyer les principes de "calculabilité". Les concept "d'interpréteur" et de "décidabilité" sont aussi exposés.

La dernière section comporte des exercices corrigés pour chaque chapitre, un énoncé d'examen avec son corrigé type et un énoncé de TP avec son corrigé type pour aider le lecteur à mieux comprendre les notions et principes développés dans ce cours.

Les sources bibliographiques ont été minutieusement choisies et consultés. Ces références répondent au contenu du cours de ce module " **Logique et fondements de l'informatique 1**" destiné à la première année Master "Systèmes informatiques et cela conformément aux programmes d'enseignement du canevas proposé.

Références Bibliographiques

Références Bibliographiques

1. Arnold André and Guessarian , Irène. Mathématiques pour l'informatique. Ed science International, 2005.

2. Bournez, Olivier. Fondements de l'informatique, Logique, modèles, et calculs, Cours INF423 de l'Ecole Polytechnique 2021.

3. Carton, Olivier. Langages formels, calculabilité et complexité. Vol. 28. Vuibert, 2008.

4. Chartrand Gary, Polimeni Albert D. and Zhang Ping. Mathematical Proofs A Transition to Advanced Mathematics, Fourth Edition Pearson, New York, 2018.

5. Dehornoy Patrick. La théorie des ensembles. Calvage et Mounet, 2017.

6. Dowek Gilles, Les démonstrations et les algorithmes : introduction à la logique et à la calculabilité, Editions de l'Ecole Polytechnique, 2010.

7. Herbet B Enderton, A Mathematical Introduction to Logic, Edition 2, Harcourt Academic Press, New York, 2013.

8. Hopcroft, J. E., Motwani, R., and Ullman, J. D. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, 2nd edition, 2001.

9. Lassaigne, Richard, Michel de Rougemont. Logique et fondements de l'informatique Hermes1993.

10. Leary Christopher C., A Friendly Introduction to Mathematical Logic, Prentice Hall, New Jersey, 2016.

11. Prost Frederic. Notes de cours Modèles de Calcul, Machine de Turing, L3 - Informatique 2021.

12. Stern, Jacques. Fondements mathématiques de l'informatique. Ediscience International, Paris, 1994.

13. Velleman Daniel J., How to prove it, second edition, Cambridge university press, Cambridge 2006.

14. Wolper, Pierre, Introduction à la calculabilité : cours et exercices corrigés. Dunod, 2001.